digital

VAX-11 COBOL-74

User's Guide

Order No. AA-C986A-TE

VAX11

$ COBOL/C74/LIST mac.
$ LINK mac, SYS$LIBRARY:C74LIB/LIB

January 1979

This document describes how to use the VAX-11 COBOL-74 compiler.

VAX-11 COBOL-74

User's Guide

Order No. AA-C986A-TE

OPERATING SYSTEM AND VERSION: VAX/VMS V01.5

SOFTWARE VERSION: VAX-11 COBOL-74 V04

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by Digital.

Copyright © 1979 by Digital Equipment Corporation

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	

CONTENTS

			Page
PREFACE			xiii
ACKNOWLEDGMENTS			xiv
CHAPTER	1	INTRODUCTION	1-1
CHAPTER	2	USING THE VAX-11 COBOL-74 SYSTEM	2-1
	2.1	CREATING A SOURCE FILE	2-1
	2.1.1	Choosing a Reference Format	2-1
	2.1.2	Entering a Source Program	2-2
	2.2	USING THE COMPILER	2-2
	2.2.1	The Command Line Format	2-2
	2.2.2	Command Qualifiers	2-3
	2.2.3	Error Message Summary	2-6
	2.2.4	Common COBOL-74 Command Line Errors	2-6
	2.3	LINKING COBOL-74 PROGRAMS	2-6
	2.4	EXECUTING A COBOL IMAGE	2-12
	2.4.1	Setting and Resetting Program Switches	2-12
	2.4.2	The RUN Command	2-13
CHAPTER	3	NON-NUMERIC DATA HANDLING	3-1
	3.1	INTRODUCTION	3-1
	3.2	DATA ORGANIZATION	3-2
	3.2.1	Group Items	3-2
	3.2.2	Elementary Items	3-2
	3.3	SPECIAL CHARACTERS	3-3
	3.4	TESTING NON-NUMERIC FIELDS	3-4
	3.4.1	Relation Tests	3-4
	3.4.1.1	Classes of Data	3-5
	3.4.1.2	The Comparison Operation	3-6
	3.4.2	Class Tests	3-7
	3.5	DATA MOVEMENT	3-7
	3.6	THE MOVE STATEMENT	3-8
	3.6.1	Group Moves	3-9
	3.6.2	Elementary Moves	3-9
	3.6.2.1	Edited Moves	3-10
	3.6.2.2	Justified Moves	3-11
	3.6.3	Multiple Receiving Fields	3-12
	3.6.4	Subscripted Moves	3-12
	3.6.5	Common Errors, MOVE Statement	3-13
	3.6.6	Format 2, MOVE CORRESPONDING	3-13
	3.7	THE STRING STATEMENT	3-14
	3.7.1	Multiple Sending Fields	3-14
	3.7.2	The POINTER Phrase	3-15
	3.7.3	The DELIMITED BY Phrase	3-16
	3.7.4	The OVERFLOW Phrase	3-18
	3.7.5	Subscripted Fields in STRING Statements	3-20
	3.7.6	Common Errors, STRING Statement	3-22
	3.8	THE UNSTRING STATEMENT	3-22
	3.8.1	Multiple Receiving Fields	3-23

CONTENTS (Continued)

			Page
	3.8.2	The DELIMITED BY Phrase	3-25
	3.8.2.1	Multiple Delimiters	3-29
	3.8.3	The COUNT Phrase	3-30
	3.8.4	The DELIMITER Phrase	3-31
	3.8.5	The POINTER Phrase	3-32
	3.8.6	The TALLYING Phrase	3-34
	3.8.7	The OVERFLOW Phrase	3-36
	3.8.8	Subscripted Fields in UNSTRING Statements	3-37
	3.8.9	Common Errors, UNSTRING Statement	3-39
	3.9	THE INSPECT STATEMENT	3-39
	3.9.1	The BEFORE/AFTER Phrase	3-40
	3.9.2	Implicit Redefinition	3-42
	3.9.3	The INSPECT Operation	3-43
	3.9.3.1	Setting the Scanner	3-45
	3.9.3.2	Active/Inactive Arguments	3-45
	3.9.3.3	Finding an Argument Match	3-46
	3.9.4	Subscripted Fields in INSPECT Statements	3-47
	3.9.5	The TALLYING Phrase	3-48
	3.9.5.1	The Tally Counter	3-48
	3.9.5.2	The Tally Argument	3-48
	3.9.5.3	The Tally Argument List	3-50
	3.9.5.4	Interference in Tally Argument Lists	3-51
	3.9.6	The REPLACING Phrase	3-55
	3.9.6.1	The Search Argument	3-56
	3.9.6.2	The Replacement Value	3-57
	3.9.6.3	The Replacement Argument	3-58
	3.9.6.4	The Replacement Argument List	3-58
	3.9.6.5	Interference in Replacement Argument Lists	3-60
	3.9.7	Common Errors, INSPECT Statement	3-60
CHAPTER	4	NUMERIC CHARACTER HANDLING	4-1
	4.1	USAGES	4-1
	4.1.1	DISPLAY	4-1
	4.1.2	COMPUTATIONAL	4-1
	4.1.3	COMPUTATIONAL-3	4-2
	4.2	DECIMAL SCALING POSITION	4-3
	4.3	SIGN CONVENTIONS	4-4
	4.4	ILLEGAL VALUES IN NUMERIC FIELDS	4-5
	4.5	TESTING NUMERIC FIELDS	4-6
	4.5.1	Relation Tests	4-6
	4.5.2	Sign Tests	4-7
	4.5.3	Class Tests	4-7
	4.6	THE MOVE STATEMENT	4-8
	4.6.1	Group Moves	4-8
	4.6.2	Elementary Numeric Moves	4-9
	4.6.3	Elementary Numeric Edited Moves	4-10
	4.6.4	Common Errors, Numeric MOVE Statements	4-12
	4.7	THE ARITHMETIC STATEMENTS	4-13
	4.7.1	Intermediate Results	4-13
	4.7.2	The ROUNDED Phrase	4-14
	4.7.3	The SIZE ERROR Phrase	4-15

CONTENTS (Continued)

			Page
	4.7.4	The GIVING Phrase	4-16
	4.7.5	Multiple Operands in ADD and SUBTRACT Statements	4-16
	4.7.6	The ADD Statement	4-17
	4.7.7	The SUBTRACT Statement	4-18
	4.7.8	The MULTIPLY Statement	4-18
	4.7.9	The DIVIDE Statement	4-19
	4.7.10	The COMPUTE Statement	4-20
	4.7.11	Common Errors, Arithmetic Statements	4-20
	4.8	ARITHMETIC EXPRESSION PROCESSING	4-21
CHAPTER	5	TABLE HANDLING	5-1
	5.1	INTRODUCTION	5-1
	5.2	DEFINING TABLES	5-1
	5.2.1	The OCCURS Phrase - Format 1	5-2
	5.2.2	The OCCURS Phrase - Format 2	5-3
	5.3	MAPPING TABLE ELEMENTS	5-3
	5.3.1	Initializing Tables	5-7
	5.4	SUBSCRIPTING AND INDEXING	5-9
	5.4.1	Subscripting with Literals	5-10
	5.4.2	Operations Performed by the Software	5-11
	5.4.3	Subscripting with Data-Names	5-12
	5.4.4	Operations Performed by the RTS	5-12
	5.4.5	Subscripting with Indexes	5-13
	5.4.6	Operations Performed by the RTS	5-14
	5.4.7	Relative Indexing	5-14
	5.4.8	Index Data Items	5-15
	5.4.9	The SET Statement	5-16
	5.4.10	Referencing a Variable-Length Table Element at RTS Time	5-17
	5.4.11	Referencing a Dynamic Group at RTS Time	5-17
	5.4.12	The SEARCH Verb	5-17
	5.4.13	The SEARCH Verb - Format 1	5-18
	5.4.14	The SEARCH Verb - Format 2	5-19
CHAPTER	6	INPUT-OUTPUT PROCESSING	6-1
	6.1	RECORD FORMAT	6-2
	6.1.1	Fixed-length	6-2
	6.1.2	Variable-length	6-3
	6.1.3	Variable with Fixed-length Control	6-3
	6.2	RECORD SIZE	6-4
	6.3	RECORD BLOCKING	6-5
	6.3.1	Sequential Files on Magnetic Tape	6-6
	6.3.2	Sequential Files on Disk	6-7
	6.3.3	Relative Files	6-8
	6.3.4	Indexed Files	6-9
	6.4	CURRENT RECORD AREA	6-10
	6.4.1	Effects on Output Operations	6-10
	6.4.2	Effects of Input Operations	6-11
	6.4.3	Sharing Record Areas	6-11

CONTENTS (Continued)

			Page
	6.5	I/O BUFFERS	6-13
	6.5.1	RMS Buffer Defaults	6-13
	6.5.2	Multiple Buffers (RESERVE Clause)	6-13
	6.5.3	Sharing Buffers (SAME AREA Clause)	6-14
	6.6	OPENING FILES	6-14
	6.6.1	I/O Operations	6-14
	6.6.2	OPEN Statement Execution	6-16
	6.7	NAMING FILES	6-17
	6.7.1	File Specifications	6-17
	6.7.2	Logical Names	6-19
	6.7.3	ASSIGN and VALUE OF ID Clauses	6-20
	6.7.4	File Switches (PDP-11 COBOL Compatibility)	6-22
	6.8	FILE COMPATIBILITY	6-24
	6.8.1	Data Type Differences	6-24
	6.8.2	Data Record Formatting Differences	6-25
	6.8.3	Special Control Characters	6-25
	6.9	I/O ERROR PROCESSING	6-25
	6.10	LOW-VOLUME I/O (ACCEPT AND DISPLAY)	6-26
	6.10.1	Mnemonic-Names (SPECIAL-NAMES Paragraph)	6-27
	6.10.2	Logical Name "Devices"	6-27
	6.10.3	ACCEPT Statement	6-28
	6.10.4	DISPLAY Statement	6-29
CHAPTER	7	GOOD PROGRAMMING PRACTICES	7-1
	7.1	FORMATTING THE SOURCE PROGRAM	7-1
	7.2	USE OF PUNCTUATION	7-5
	7.3	USE OF THE ALTER STATEMENT	7-5
	7.4	USE OF THE PERFORM STATEMENT	7-6
	7.5	USE OF LEVEL-88 CONDITION NAMES	7-7
	7.6	USE OF QUALIFIED REFERENCES	7-9
	7.6.1	Qualified Data References	7-9
	7.6.2	Guideline 1 (Data Item Definition)	7-12
	7.6.3	Guideline 2 (Reference Format)	7-12
	7.6.4	Guideline 3 (Unique Referability)	7-13
	7.6.5	Qualified Procedure References	7-13
	7.6.6	Qualification and Compiler Performance	7-13
CHAPTER	8	REFORMAT UTILITY PROGRAM	8-1
CHAPTER	9	DEBUGGING COBOL PROGRAMS	9-1
	9.1	DEBUG CONCEPTS	9-1
	9.2	PREPARING TO DEBUG A PROGRAM	9-2
	9.2.1	SET LANGUAGE COBOL Command	9-2
	9.2.2	MODULE Commands: SET, SHOW, and CANCEL	9-2
	9.2.3	SCOPE Commands: SET, SHOW, and CANCEL	9-3
	9.3	SPECIFYING LOCATIONS	9-4
	9.3.1	Location Types	9-4
	9.3.2	Resolving Location Ambiguities	9-5

CONTENTS (Continued)

			Page
	9.4	CONTROLLING PROGRAM EXECUTION	9-5
	9.4.1	BREAK Commands: SET, SHOW, and CANCEL	9-5
	9.4.2	TRACE Commands: SET, SHOW, and CANCEL	9-6
	9.4.3	WATCH Commands: SET, SHOW, and CANCEL	9-7
	9.4.4	GO and STEP Commands	9-8
	9.4.5	CTRL/Y Command (Interrupting the Image)	9-9
	9.4.6	EXIT Command	9-9
	9.4.7	SHOW CALLS Command	9-9
	9.5	EXAMINING AND CHANGING DATA	9-10
	9.5.1	EXAMINE Command	9-10
	9.5.2	DEPOSIT Command	9-10
	9.6	SAMPLE DEBUG SESSION	9-11
CHAPTER	10	ERROR MESSAGES	10-1
	10.1	COMPILE-TIME ERROR MESSAGES	10-1
	10.1.1	Severity Levels	10-2
	10.1.2	Error Message Printing	10-3
	10.1.3	Internal Compiler Errors -- System Errors	10-3
	10.2	SYSTEM MESSAGES	10-4
	10.2.1	Link-Time Error Messages	10-5
	10.2.2	Run-Time Error Messages	10-5
	10.2.2.1	Faulty Program Logic Error Procedures	10-5
	10.2.2.2	File I/O Error Procedures	10-7
CHAPTER	11	SORTING IN A COBOL PROGRAM	11-1
	11.1	VAX-11 SORT SUBROUTINE PACKAGE	11-1
	11.2	I/O INTERFACE METHODS	11-2
	11.2.1	File I/O Interface	11-2
	11.2.2	Record I/O Interface	11-2
	11.3	KEY DATA AND RECORD AREAS	11-3
	11.4	KEY BUFFER	11-4
	11.5	SORT SUBROUTINES	11-6
	11.5.1	SOR$PASS_FILES	11-6
	11.5.2	SOR$INIT_SORT	11-7
	11.5.3	SOR$RELEASE_REC	11-8
	11.5.4	SOR$SORT_MERGE	11-9
	11.5.5	SOR$RETURN_REC	11-10
	11.5.6	SOR$END_SORT	11-11
	11.6	PROGRAMMING EXAMPLE	11-12
CHAPTER	12	USING THE LIBRARY FACILITY	12-1
	12.1	Creating a COBOL Library File	12-2
	12.2	The COPY Statement	12-2
	12.3	The COPY REPLACING Statement	12-4
	12.4	The Source Listing	12-6
	12.5	Common Errors In Using the Library Facility	12-7

CONTENTS (Continued)

			Page
CHAPTER	13	OPTIMIZATION	13-1
	13.1	OPTIMIZING FILE DESIGN	13-2
	13.1.1	Sequential Files	13-2
	13.1.2	Relative Files	13-2
	13.1.3	Indexed Files	13-3
	13.1.3.1	General Rules for Indexed Files	13-5
	13.1.3.2	Bucket Size	13-6
	13.1.3.3	Index Depth	13-7
	13.1.3.4	Overhead Accumulation	13-7
	13.2	OPTIMIZING PROGRAM ORGANIZATION	13-8
	13.2.1	Sequential Reading of Indexed Files	13-8
	13.2.2	Caching Index Roots	13-8
	13.2.3	Multi-block Reading and Writing	13-9
	13.3	OPTIMIZING COMPUTATION	13-9
APPENDIX	A	THE COBOL FORMATS	A-1
APPENDIX	B	COMPILER IMPLEMENTATION LIMITATIONS	B-1
APPENDIX	C	SOURCE PROGRAM LISTINGS	C-1
APPENDIX	D	DIAGNOSTIC ERROR MESSAGES	D-1
APPENDIX	E	RUN-TIME ERROR MESSAGES	E-1
APPENDIX	F	INTERNAL COMPILER ERRORS -- SYSTEM ERRORS	F-1
APPENDIX	G	PROGRAMMING EXAMPLES	G-1
	G.1	CALLING A FORTRAN SUBROUTINE	G-1
	G.1.1	The COBOL Program, GETROOT	G-1
	G.1.2	The FORTRAN Program, SQROOT	G-3
	G.1.3	Sample Run of GETROOT	G-3
	G.2	CALLING VAX-11 RUN-TIME PROCEDURES	G-4
	G.2.1	The COBOL Program, RUNTIME	G-4
	G.2.2	Sample Run of RUNTIME	G-5
	G.3	USING TERMINAL ESCAPE SEQUENCES	G-5
	G.3.1	The COBOL Program, ESCAPE	G-6
	G.3.2	Sample Run of ESCAPE	G-9
	G.4	CALLING VAX/VMS SYSTEM SERVICES	G-10
	G.4.1	The COBOL Program, SYSTSVC	G-10
	G.4.2	Sample Run of SYSTSVC	G-13
INDEX			Index-1

FIGURES

FIGURE	1-1	Building a COBOL Task Image	1-2
	3-1	Field Sizes	3-3
	3-2	Redefining Special Characters	3-4

			Page
FIGURE	3-3	Relation Condition	3-5
	3-4	The Meanings of Relational Operators	3-5
	3-5	Class Condition, General Format	3-7
	3-6	Data Movement with Editing Symbols	3-11
	3-7	Data Movement with No Editing	3-11
	3-8	Subscripted MOVE Statements	3-12
	3-9	Sample STRING Statement	3-14
	3-10	Concatenation with the STRING Statement	3-14
	3-11	Literals as Sending Fields	3-15
	3-12	Indexed Sending Fields	3-15
	3-13	Sample POINTER Phrase	3-16
	3-14	Delimiting with the Word SIZE	3-16
	3-15	SPACE as a Delimiter	3-17
	3-16	Repeating the DELIMITED BY Phrase	3-17
	3-17	Delimiting with More Than One Space Character	3-18
	3-18	The ON OVERFLOW Phrase	3-18
	3-19	Various STRING Statements Illustrating the Overflow Condition	3-19
	3-20	STRING Statement with Pointer	3-20
	3-21	Subscripting with the Pointer	3-20
	3-22	Subscripting with the Delimiter	3-21
	3-23	Sample UNSTRING Statement	3-22
	3-24	Multiple Receiving Fields	3-23
	3-25	Delimiting with a Space Character	3-25
	3-26	Delimiting with Multiple Receiving Fields	3-26
	3-27	Delimiting with an Identifier	3-29
	3-28	Multiple Delimiters	3-29
	3-29	The COUNT Phrase	3-30
	3-30	The DELIMITER Phrase	3-31
	3-31	The POINTER Phrase	3-33
	3-32	Examining the Next Character by Using the Pointer Data Item as a Subscript	3-33
	3-33	Examining the Next Character by Placing It Into a One-Character Field	3-34
	3-34	The TALLYING Phrase	3-34
	3-35	The POINTER and TALLYING Phrases Used Together	3-35
	3-36	Subscripting the COUNT Phrase with the TALLYING Data Item	3-36
	3-37	Using the OVERFLOW Phrase	3-36
	3-38	Sequence of Subscript Evaluation	3-38
	3-39	Erroneously Repeating the Word INTO	3-39
	3-40	Sample INSPECT...TALLYING Statement	3-40
	3-41	Sample INSPECT...REPLACING Statement	3-40
	3-42	Sample INSPECT...BEFORE Statement	3-40
	3-43	Matching the Delimiter Characters to the Characters in a Field	3-41
	3-44	Sample INSPECT Statement	3-44
	3-45	Sample REPLACING Argument	3-44
	3-46	Sample AFTER Delimiter Phrase	3-45
	3-47	Where Arguments Become Active in a Field	3-46

			Page
FIGURE	3-48	Sample Subscripted Argument	3-47
	3-49	Format of the Tally Argument	3-48
	3-50	CHARACTERS Form of the Tally Argument	3-48
	3-51	Results of Counting with the LEADING Condition	3-49
	3-52	Argument List Adding into One Tally Counter	3-50
	3-53	Argument List Adding into Separate Tally Counters	3-50
	3-54	Argument List (with Delimiters) Adding into Separate Tally Counters	3-50
	3-55	Results of the Scan in Figure 3-55	3-51
	3-56	Two Tallying Arguments that Do Not Interfere with Each Other	3-51
	3-57	Two Tallying Arguments that Do Interfere with Each Other	3-52
	3-58	Two Tallying Arguments that, Because of Their Positioning, Only Partially Interfere with Each Other	3-52
	3-59	An Attempt to Tally the Character B with Two Arguments	3-52
	3-60	Tallying Asterisk Groupings	3-53
	3-61	Placing the LEADING Condition in the Argument List	3-53
	3-62	Reversing the Argument List in Figure 3-62	3-54
	3-63	An Argument List that Counts Words in a Statement	3-54
	3-64	Counting Leading Tab or Space Characters	3-55
	3-65	Counting the Remaining Characters with the CHARACTERS Argument	3-55
	3-66	Format of the Search Argument	3-56
	3-67	Format of the Replacement Value	3-57
	3-68	The Replacement Argument	3-58
	3-69	Replacement Argument List that is Active Over the Entire Field	3-58
	3-70	Replacement Argument List that "Swaps" Ones for Zeroes and Zeroes for Ones	3-58
	3-71	Replacement Argument List that Becomes Inactive with the Occurrence of a Space Character	3-59
	3-72	Argument List with Three Arguments that Become Inactive with the Occurrence of a Space	3-59
	4-1	Memory Storage of COMP Data Items	4-2
	4-2	Memory Storage of COMP-3 Data Items	4-3
	4-3	Truncation Caused by Decimal Point Alignment	4-9
	4-4	Zero Filling Caused by Decimal Point Alignment	4-10
	4-5	Numeric Editing	4-12
	4-6	Rounding Truncated Decimal Point Positions	4-14
	4-7	Rounding Truncated Decimal Scaling Positions	4-15

			Page
FIGURE	4-8	Explicit Programmer-Defined Temporary Work Area	4-21
	4-9	Arithmetic Statement Intermediate Result Field Attributes Determined from Composite of Operands	4-22
	4-10	Arithmetic Expression Intermediate Result Field Attributes Determined by Implementor-Defined Rules	4-22
	5-1	Defining a Table	5-2
	5-2	Mapping a Table into Memory	5-3
	5-3	Synchronized COMP Item in a Table	5-4
	5-4	Adding a Field without Altering the Table Size	5-5
	5-5	Adding One Byte which Adds Two Bytes to the Element Length	5-5
	5-6	Forcing an Odd Address by Adding a 1-Byte FILLER Item to the Head of the Table	5-6
	5-7	The Effect of a SYNCHRONIZED RIGHT Clause Instead of a FILLER Item as Shown in Figure 5-6	5-7
	5-8	Initializing Tables	5-8
	5-9	Initializing Mixed Usage Fields	5-8
	5-10	Initializing Alphanumeric Fields	5-9
	5-11	Literal Subscripting	5-10
	5-12	Subscripting a Multi-Dimensional Table	5-10
	5-13	Subscripting Rules for a Multi-Dimensional Table	5-11
	5-14	Subscripting with Data-Names	5-12
	5-15	Index-Name Item	5-13
	5-16	Subscripting with Index-Name Items	5-13
	5-17	Relative Indexing	5-15
	5-18	Index Data Item	5-16
	5-19	Legal Data Movement with the SET Statement	5-16
	5-20	Example of Using SEARCH to Search a Table	5-21
	7-1	Unqualified Data Item Reference	7-10
	7-2	Qualified Data Item Reference	7-11
	7-3	General Format of a Qualified Data Reference	7-12
	7-4	General Format of a Qualified Procedure Reference	7-13
	12-1	Merging Library Text	12-3
	13-1	Three-Level Primary Key Index	13-4

TABLES

TABLE	2-1	Command Qualifiers	2-3
	3-1	Legal Non-Numeric Elementary Moves	3-9
	3-2	Results of the Preceding Sample Statements	3-19
	3-3	Results of the Preceding Sample Statements	3-21
	3-4	Values Moved into the Receiving Fields Based on the Value in the Sending Field	3-24
	3-5	Handling a Sending Field that is Too Short	3-25

			Page
TABLE	3-6	Results of Delimiting with an Asterisk	3-26
	3-7	Results of Delimiting Multiple Receiving Fields	3-27
	3-8	Results of Delimiting with Two Asterisks	3-27
	3-9	Results of Delimiting with ALL Asterisks	3-28
	3-10	Results of Delimiting with ALL Double Asterisks	3-28
	3-11	Results of the Multiple Delimiters Shown in Figure 3-29	3-30
	3-12	Original, Altered, and Restored Values Resulting from Implicit Redefinition	3-43
	4-1	The Resulting ASCII Character from a Sign and Digit Sharing the Same Byte	4-5
	4-2	The Sign Tests	4-7
	6-1	I/O Statements Grouped by File Organization, Access Mode, and Open Mode	6-15
	6-2	File Specification Switches for PDP-11 Compatibility	6-22
	12-1	COPY REPLACING Matches	12-6

Commercial Engineering Publications typeset this manual using DIGITAL's TMS-11 System.

936LL

PREFACE

MANUAL OBJECTIVES

This manual describes the VAX-11 COBOL-74 compiler. It discusses the relationships between the COBOL-74 language, the compiler, object modules and executable images, and VAX/VMS and its utilities. The User's Guide supplements the description of the COBOL-74 programming language in the VAX-11 COBOL-74 Language Reference Manual.

INTENDED AUDIENCE

This manual is designed for programmers who have a working knowledge of the COBOL-74 language and who are familiar with the basic concepts of VAX/VMS.

STRUCTURE OF THIS DOCUMENT

The User's Guide is organized into chapters and appendixes that describe functions, concepts, and features of the VAX-11 COBOL-74 language system.

ASSOCIATED DOCUMENTS

This manual refers to the following documents, which contain supplemental information that is relevant to VAX-11 COBOL-74 programming:

- VAX-11 COBOL-74 Language Reference Manual
- VAX/VMS Command Language User's Guide
- VAX-11 Linker Reference Manual
- Introduction To VAX-11 Record Management Services
- VAX-11 Symbolic Debugger Reference Manual
- VAX-11 Sort Reference Manual
- VAX/VMS Operator's Guide

CONVENTIONS USED IN THIS DOCUMENT

The syntactic conventions used in general format examples are discussed in Chapter 1 of the VAX-11 COBOL-74 Language Reference Manual.

ACKNOWLEDGMENT

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted material used herein are: FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the UNIVAC (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell.

They have specifically authorized the use of this material, in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

Procedures have been established for the maintenance of COBOL. Inquiries concerning the procedures for proposing changes should be directed to the Chairman of the CODASYL COBOL Committee, P.O. Box 1808, Washington, DC 20013.

CHAPTER 1

INTRODUCTION

The VAX-11 COBOL-74 compiler translates ANS-74 COBOL source programs into relocatable object modules; it runs under the supervision of VAX/VMS.

To run a COBOL program, you follow a four-step process:

- Prepare a source program
- Compile a source program
- Link object modules into an executable image file
- Execute the image

The VAX-11 COBOL-74 compiler accepts COBOL source statements from source input files. This means that you must manually enter your source statements onto an acceptable medium prior to the compilation process.

Once you have decided upon an input medium and format for your source input files and have created them, you compile the source program. The VAX-11 COBOL-74 compiler reads source statements from the source input file and translates them into an object module consisting of program sections (PSECTs). It can also produce a source program listing with optional special-purpose listings, such as a map and cross-reference. Chapter 2 describes the procedure for compiling programs and specifying compiler options.

The compiler can compile only one source program or subprogram at a time. Therefore, a program that consists of a main program and one or more subprograms requires multiple executions of the compiler. Each compilation generates a separate listing and object module.

The compiler produces an object module, which must be linked by the VAX-11 Linker to produce an executable image file. The linker can combine several independently compiled object modules into a single executable image; the ability to compile COBOL subprograms to produce linkable object modules enables you to create modular programs.

The image is an executable form of the declarations and instructions in your COBOL source programs. It includes subprograms that were included by the linker as a result of your commands. It also includes routines from the COBOL run-time system (RTS), which is a library of predefined generalized procedures that perform standard functions for your program.

Figure 1-1 shows the process of preparing a COBOL program for execution.

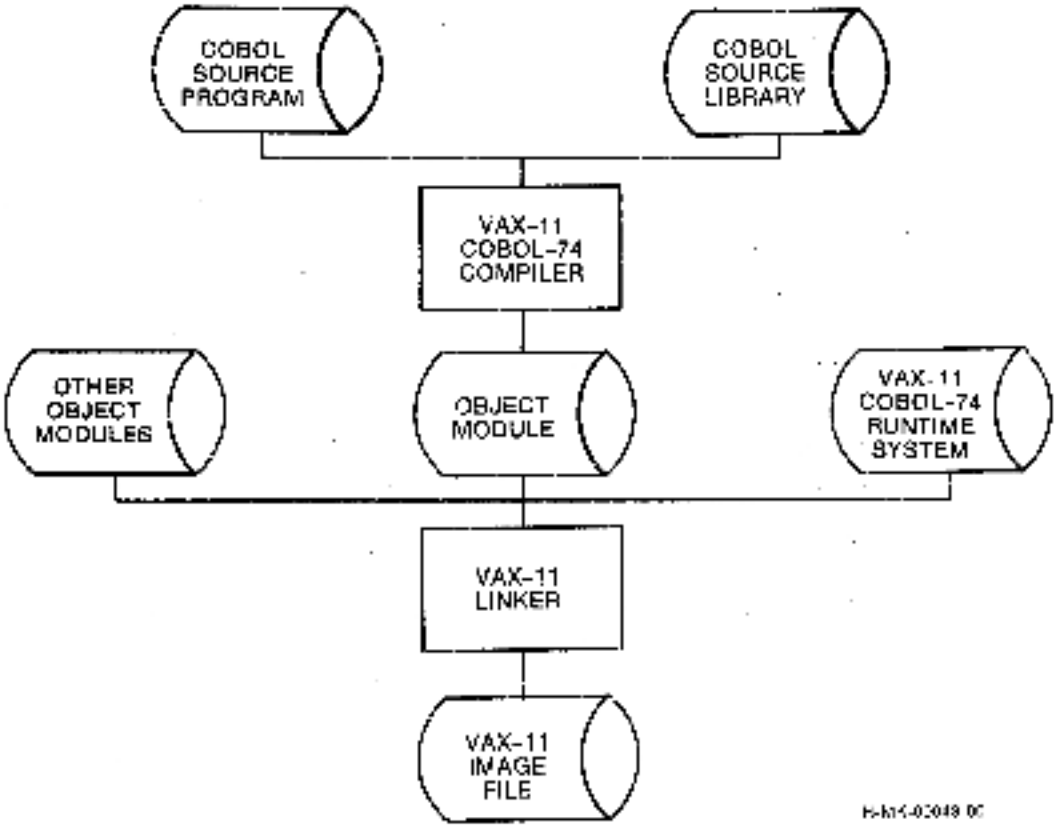


Figure 1-1
Building an Executable Image

CHAPTER 2

USING THE VAX-11 COBOL-74 SYSTEM

This chapter discusses the procedures for creating, compiling, linking, and executing COBOL programs.

2.1 CREATING A SOURCE PROGRAM

Before you can compile a COBOL program, you must decide on the source reference format and prepare your source program for input to the compiler.

2.1.1 Choosing a Reference Format

The VAX-11 COBOL-74 compiler can accept source programs in either conventional or terminal reference format (both are described in the VAX-11 COBOL-74 Reference Manual). However, you cannot mix reference formats in the same source program (including text copied from a COBOL library).

Terminal format was designed to be easily used by programmers at interactive terminals; therefore, the compiler accepts terminal reference format as a default and allows you to use a command qualifier to specify conventional format. The terminal format can reduce the amount of file space needed to store source programs. In addition, it is usually easier to edit source programs written in terminal format, because spacing requirements are more flexible.

You may want to select the conventional reference format, however, if your COBOL program was originally written that way for another compiler.

You can convert a terminal format program to conventional format by using the REFORMAT utility, which is described in Chapter 8. You can also use REFORMAT to match the formats of source files and COBOL library files if they are not the same.

2.1.2 Entering a Source Program

You can create a source program file by using the VAX/VMS CREATE command or a text editor. CREATE can be used only for a new file; you must use a text editor to change existing source files. Most users rely on text editors for both creating and updating source files.

Unless you specify a file type for the source program file in the command line, which is described in the next section, the compiler assumes COB as a default; therefore, you can simplify compiling by naming source files with the default file type.

The CREATE command is described in the VAX/VMS Command Language User's Guide; the VAX/VMS Text Editing Reference Manual discusses the SOS and SLP text editors.

2.2 USING THE COMPILER

The VAX-11 COBOL-74 compiler translates source statements into object modules that contain relocatable code. It can also produce a listing of source statements and other information if you use the appropriate command qualifiers. This section describes the procedure for compiling your source program; it discusses the COBOL command line and the error message summary. Finally, it lists some common errors to avoid in entering compiler command lines. Appendix C discusses the components of the source program listing.

2.2.1 The Command Line Format

The VAX-11 COBOL-74 command line has the following format:

```
COBOL/C74 [/command-qualifiers] file-spec
```

where:

COBOL/C74	specifies the VAX-11 COBOL-74 compiler.
/command-qualifiers	specify compiler options.
file-spec	specifies the file that contains the COBOL source program. If you do not supply a file type in the file specification, the compiler uses COB as the default. Do not use wild cards in the file specification.

2.2.2 Command Qualifiers

VAX-11 COBOL-74 provides a series of command qualifiers that you can use to select or suppress compiler options. Table 2-1 summarizes the qualifiers, which are then described in detail.

Table 2-1
Command Qualifiers

Qualifier	Default
/[NO]ANSI_FORMAT	/NOANSI_FORMAT
/[NO]COPY_LIST	/COPY_LIST
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG	/NODEBUG
/[NO]LIST[=file-spec]	
/[NO]MAP	/NOMAP
/[NO]OBJECT[=file-spec]	
/[NO]VERB_LOCATION	/NOVERB_LOCATION
/[NO]WARNINGS	/WARNINGS

/ANSI_FORMAT
/NOANSI_FORMAT
Indicates whether the source program is in ANSI (conventional) format or in DIGITAL's terminal format.

For conventional format, the compiler expects 80-character images with optional sequence numbers in character positions 1-6, indicators in position 7, Area A beginning in position 8, Area B beginning in position 12, and the identification area in positions 73-80.

By default, the compiler assumes that the source file is in terminal format; that is, Area A begins in record position 1.

/COPY_LIST
/NOCOPY_LIST
Controls whether statements included by COPY statements in the source program are printed in the listing file.

/COPY_LIST is the default: the compiler includes all source statements in the source listing.

/NOCOPY_LIST suppresses the listing of text copied from library files; only the COPY statement appears in the listing file.

/CROSS_REFERENCE
/NOCROSS_REFERENCE
Controls whether the source program listing includes a cross-reference listing.

/CROSS_REFERENCE produces a cross-reference listing as part of the listing file. The compiler sorts data-names and procedure-names into ascending order and lists them with the source program line numbers on which they appear. On the listing, the symbol # indicates the source line on which the name is defined. Note that the use of /CROSS_REFERENCE significantly slows down the compilation of large programs.

By default, the compiler does not create a cross-reference listing.

/DEBUG[=TRACEBACK]
/NODEBUG

Controls whether the compiler produces traceback information and local symbol table information for the debugger.

/DEBUG allows you to refer to data items by data-name, and to Procedure Division locations by line number; it generates both traceback and symbol table information. /DEBUG=TRACEBACK produces traceback information only; /NODEBUG generates neither. The default is /NODEBUG.

Chapter 9 discusses COBOL program debugging using the VAX/VMS Symbolic Debugger.

/LIST[=file-spec]
/NOLIST

Controls whether the compiler produces an output listing.

If you use the COBOL/C74 command in interactive mode, the compiler, by default, does not create a listing file.

If the COBOL/C74 command is executed from a batch job, /LIST is the default.

When you specify /LIST, you can control the defaults applied to the output file specification by where you place the qualifier in the command, as described in the VAX/VMS Command Language User's Guide.

The output file type always defaults to LIS.

/MAP
/NOMAP

/MAP causes the compiler to produce the following reports in the listing file:

- Data Division Map
- Procedure Map
- External Subprograms Referenced
- Data and Control PSECTs
- RTS Routines Referenced
- Segmentation Map

/NOMAP is the default.

/OBJECT[=file-spec]
/NOOBJECT
Controls whether the compiler produces an object file.

By default, the compiler produces an object file with the same file name as the input file and a file type of OBJ. The compiler also uses the default file type of OBJ when you include a file specification with the /OBJECT qualifier that does not have a file type.

/VERB_LOCATION
/NOVERB_LOCATION
Indicates whether the output listing produced by the compiler shows the object location of each verb in the source program.

The location appears on the line before the source line in which the verb is used.

The default is /NOVERB_LOCATION.

/WARNINGS
/NOWARNINGS
Controls whether the compiler prints informational diagnostic messages as well as warning and fatal diagnostic messages. By default, the compiler prints informational diagnostics; specify /NOWARNINGS to suppress them.

Consider the following command line examples:

```
COBOL/C74/DEBUG PROGA
```

Produces an object module file PROGA.OBJ from the source file PROGA.COB.

```
COBOL/C74/LIST/DEBUG/OBJECT=TESTB A12
```

Uses the source file A12.COB to produce object module TESTB.OBJ and a source listing in file A12.LIS.

```
COBOL/C74/LIST/CROSS_REFERENCE PAYROLL
```

Uses the source file PAYROLL.COB to produce object module PAYROLL.OBJ and a source listing with cross reference in file PAYROLL.LIS.

The debugger cannot reference data items by data-name in this module because the /DEBUG qualifier is not specified.

```
COBOL/C74/LIST=RPTB.REP/DEBUG/MAP REPORTB.TXT
```

Uses the source file REPORTB.TXT to produce object module REPORTB.OBJ and a source listing with map in file RPTB.REP.

2.2.3 Error Message Summary

If the compiler detects any errors during a compilation, it displays an error message summary on the system output device. The error message summary has the following format:

```
C74 -- nnnnn ERROR(S), nnnnn FATAL
```

NOTE

If any fatal errors occur, the compiler does not generate an object file.

2.2.4 Common COBOL-74 Command Line Errors

Some common errors to avoid when entering COBOL-74 command lines are:

- Omitting the /ANSI_FORMAT qualifier for source programs that are in conventional format.
- Including contradictory qualifiers, such as /MAP without /LIST.
- Omitting version numbers from file specifications when you want to compile other than the latest version of a source file.
- Forgetting to use a file type in the file specification when you intend to use or create a file with other than the default file type.

2.3 LINKING COBOL-74 PROGRAMS

After you have compiled one or more source programs to produce object modules, you must link the object module(s) to create a program image that can then be executed. Linking resolves symbolic references in the object code and establishes absolute addresses for them. This section describes the procedure for creating executable images from object modules using the VAX/VMS LINK command. You will find further information in the VAX/VMS Command Language User's Guide and the VAX-11 Linker Reference Manual.

To link object modules, enter a LINK command in the following format:

```
LINK [/command-qualifiers] file-spec(s) [/file-qualifiers]
```

where:

/command-qualifiers	specify output file options.
file-spec	specifies the input file(s) to be linked.
/file-qualifiers	specify input file options.

You can enter multiple file specifications separated from each other by commas or plus signs (which are equivalent). Regardless of how many file specifications you specify, the LINK command produces only one executable image.

If you do not specify a file type in an input file specification, the Linker assumes default file types, depending on the nature of the file. For example, object files are assumed to have a file type of OBJ. The VAX/VMS Command Language User's Guide discusses VAX-11 Linker default file types in detail.

Default file types for output files are discussed in the VAX/VMS Command Language User's Guide. Consider the following command line:

```
LINK TESTA,TESTB,SYS$LIBRARY:C74LIB/LIB
```

This line causes the compiler to use two object modules (TESTA.OBJ and TESTB.OBJ) to produce a single executable image (TESTA.EXE).

NOTE

The command line must specify the library that contains the COBOL-74 RTS. The examples in this chapter specify:

```
SYS$LIBRARY:C74LIB/LIB
```

You can also specify the optional sharable RTS, which results in a smaller image file and sharing of physical memory when two or more COBOL images run at the same time. Link with the sharable RTS by specifying:

```
SYS$LIBRARY:C74LIB/OPT
```

Before you can use this option, your system manager must install the sharable image, SYS$SYSTEM:C74LIB.EXE, as SHARED. The procedure is described in the VAX/VMS Operator's Guide.

The following discussion describes the command qualifiers and file qualifiers that you are most likely to use for linking COBOL modules. However, you will find complete discussions of all LINK command qualifiers in the references already mentioned. The following qualifiers are discussed:

Command qualifiers	Default
/BRIEF	
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG[=file-spec]	/NODEBUG
/[NO]EXECUTABLE[=file-spec]	/EXECUTABLE
/FULL	
/[NO]MAP[=file-spec]	/NOMAP
/[NO]TRACEBACK	/TRACEBACK
File qualifiers	
/INCLUDE=module-name[,...]	
/LIBRARY	
/OPTIONS	

Command Qualifiers:

/BRIEF

Produces a brief memory allocation map file. Use /BRIEF only if you also specify /MAP; /BRIEF must follow /MAP on the command line.

The brief form of the map contains:

1. A summary of the image characteristics.
2. A list of all object modules included in the image.
3. A summary of link-time performance statistics.

Example

```
LINK/MAP/BRIEF PROGA,SYS$LIBRARY:C74LIB/LIB
```

/CROSS_REFERENCE
/NOCROSS_REFERENCE

Controls whether the Linker produces a symbol cross-reference on the memory allocation map.

Use /CROSS_REFERENCE only if you also specify /MAP; /CROSS_REFERENCE must follow /MAP on the command line.

Example

```
LINK/MAP/CROSS_REFERENCE PROGA,SYS$LIBRARY:C74LIB/LIB
```

The symbol cross-reference lists each global symbol referenced in the image, its value, and all modules in the image that refer to it.

The default is /NOCROSS_REFERENCE.

/DEBUG[=file-spec]
/NODEBUG
Controls whether the Linker includes a debugger in the image.

If the object module contains local symbol table information for the Debugger, specify /DEBUG to include the information in the image as well.

You can include the optional file specification to specify a user-defined debugger; the default file type is OBJ. If you specify /DEBUG without a file specification, the default VAX/VMS Debugger is linked with the image. You will find more information on using /DEBUG in the VAX/VMS Symbolic Debugger Reference Manual.

The default is /NODEBUG.

Chapter 9 discusses COBOL program debugging.

/EXECUTABLE[=file-spec]
/NOEXECUTABLE
Controls whether the Linker creates an executable image and optionally supplies a file specification for the output image file.

By default, the Linker creates an executable image with the same file name as the first input file and a file type of EXE.

Use /NOEXECUTABLE to see the results of linking in less time than the Linker would need to create an image file.

Examples:

```
LINK/EXECUTABLE=NEWPROG.IMG/MAP PROGA,SYS$LIBRARY:C74LIB/LIB

LINK/NOEXECUTABLE/MAP PROGA,SYS$LIBRARY:C74LIB/LIB
```

/FULL
Produces a full memory allocation map listing. Use /FULL only if you also specify /MAP; /FULL must follow /MAP on the command line.

A full map listing contains:

1. All information contained in the brief listing.
2. Detailed descriptions of each program section and image section in the image file.
3. Lists of global symbols by name and value.

Example

```
LINK/MAP/NOEXEC/FULL PROGA,SYS$LIBRARY:C74LIB/LIB
```

/MAP[=file-spec]
/NOMAP
Controls whether the Linker produces a memory allocation map listing.

You can specify the file specification to name the map file; otherwise, the name of the output file is the same as the name of the first input file, with a file type of MAP.

When you specify /MAP, you can also specify /BRIEF, /FULL, or /CROSS_REFERENCE to control the contents of the map. If you specify none of these qualifiers, the map contains:

1. All the information contained in the brief listing.
2. A list of user-defined global symbols sorted by name.
3. A list of user-defined program sections.

The default is /NOMAP.

/TRACEBACK
/NOTRACEBACK
Controls whether the Linker includes traceback information in the image file.

By default, the Linker includes traceback information so the system can trace the call stack when an error occurs. If you specify /NOTRACEBACK, you will get no traceback reporting when errors occur.

If you specify /DEBUG, the Linker also assumes /TRACEBACK.

File Qualifiers

/INCLUDE=module-name[, ...]
Indicates that the associated file specification refers to an object module library (the default file type is OLB); furthermore, it causes the Linker to unconditionally include only the specified module(s).

You must specify at least one module-name. Specify more than one by separating them with commas and enclosing the list in parentheses.

You can also specify /LIBRARY when you specify /INCLUDE to cause the Linker to search the library for unresolved references after it unconditionally includes the specified module(s).

Examples:

```
LINK PROGA,LIBA/INCLUDE=MODA,SYS$LIBRARY:C74LIB/LIB
```

The Linker links PROGA.OBJ and the module MODA from the library file LIBA.OLB to produce PROGA.EXE.

```
LINK PROGA,LIBA/INC=(MODA,MODB)/LIB,SYS$LIBRARY:C74LIB/LIB
```

The Linker links PROGA.OBJ and the modules MODA and MODB from the library file LIBA.OLB. Because of the /LIBRARY file qualifier, the Linker will also search LIBA.OLB for any other unresolved references in PROGA.OBJ, MODA, and MODB.

/LIBRARY

Indicates that the file specification refers to a library file to be searched to resolve any undefined symbols in the input file(s).

If the file specification does not include a file type, the Linker assumes the default file type OLB. Do not specify a library as the first input file unless you also specify the /INCLUDE qualifier to indicate which modules in the library are to be unconditionally included in the image. You can use both /INCLUDE and /LIBRARY; this causes the Linker to include the specified modules, then search the library for unresolved references.

Examples

```
LINK PROGA,LIBA/LIBRARY,SYS$LIBRARY:C74LIB/LIB
```

The Linker searches LIBA.OLB for unresolved references in PROGA.OBJ to create PROGA.EXE.

```
LINK LIBA/LIB/INCLUDE=MOD1/EXEC=PROG,SYS$LIBRARY:C74LIB/LIB
```

The Linker includes the module MOD1 from LIBA.OLB, then searches LIBA.OLB for unresolved references in MOD1. The result is an executable image PROG.EXE.

/OPTIONS

Indicates that the input file contains a list of options to control linking. If the /OPTIONS file specification does not include a file type, the Linker uses the default file type OPT.

The contents of the option file are described in the VAX-11 Linker Reference Manual.

2.4 EXECUTING A COBOL IMAGE

When the object modules have been linked to create an executable image, you can use the RUN command to execute the image in the process. If you specified SWITCH ON or OFF in the SPECIAL-NAMES paragraph of the COBOL source program, you can specify the states of switches before or during image execution.

2.4.1 Setting and Resetting Program Switches

COBOL program switches exist as the logical name COB$SWITCHES, which can be defined for the process, group, or system. Use the DEFINE command (you can also use the ASSIGN command) to change the status of program switches:

```
DEFINE COB$SWITCHES "switch-list"
```

where switch-list is a list of one or more program switch numbers (1-16) separated by commas. The entire list must be enclosed in quotes. A switch is set ON if its number appears in the switch-list; otherwise, it is set OFF.

Examples

```
DEFINE COB$SWITCHES "1,5"
```

Sets switches 1 and 5 ON; sets all others OFF.

```
DEFINE COB$SWITCHES "4,5,6,7,8,9,10,11,12,13,14,15,16"
```

Sets all switches ON except 1, 2, and 3.

```
DEFINE COB$SWITCHES " "
```

Turns OFF all switches.

The order of evaluation of logical name assignments is: process, group, system. System and group logical name assignments (including COBOL program switch settings) continue until they are changed (or deassigned). Process logical name assignments exist until either they are changed (or deassigned) or until the process terminates. Therefore, you should be aware of system and group assignments of COB$SWITCHES before executing an image if you do not define it yourself in your process.

You can guarantee the intended status of COBOL program switches by setting switches just before executing an image that uses them. You can confirm the switch settings by using the following command:

```
SHOW LOGICAL COB$SWITCHES
```

You can use the DEASSIGN command to remove the switch-setting logical name from your process; the group or system logical name (if any) is then active:

```
DEASSIGN COB$SWITCHES
```

You can also change the status of switches during execution:

1. Interrupt the image with CTRL/Y or a STOP literal COBOL statement.
2. Use a DEFINE command to change switch settings.
3. Continue the image with a CONTINUE command. Be sure that you do not force the interrupted image to exit by entering a command that executes another image.

2.4.2 The RUN Command

Use the RUN command to execute an image:

```
RUN [/command-qualifier] file-spec
```

If you do not specify a file type in file specification, the RUN command uses the default file type EXE.

The RUN command has two optional command qualifiers:

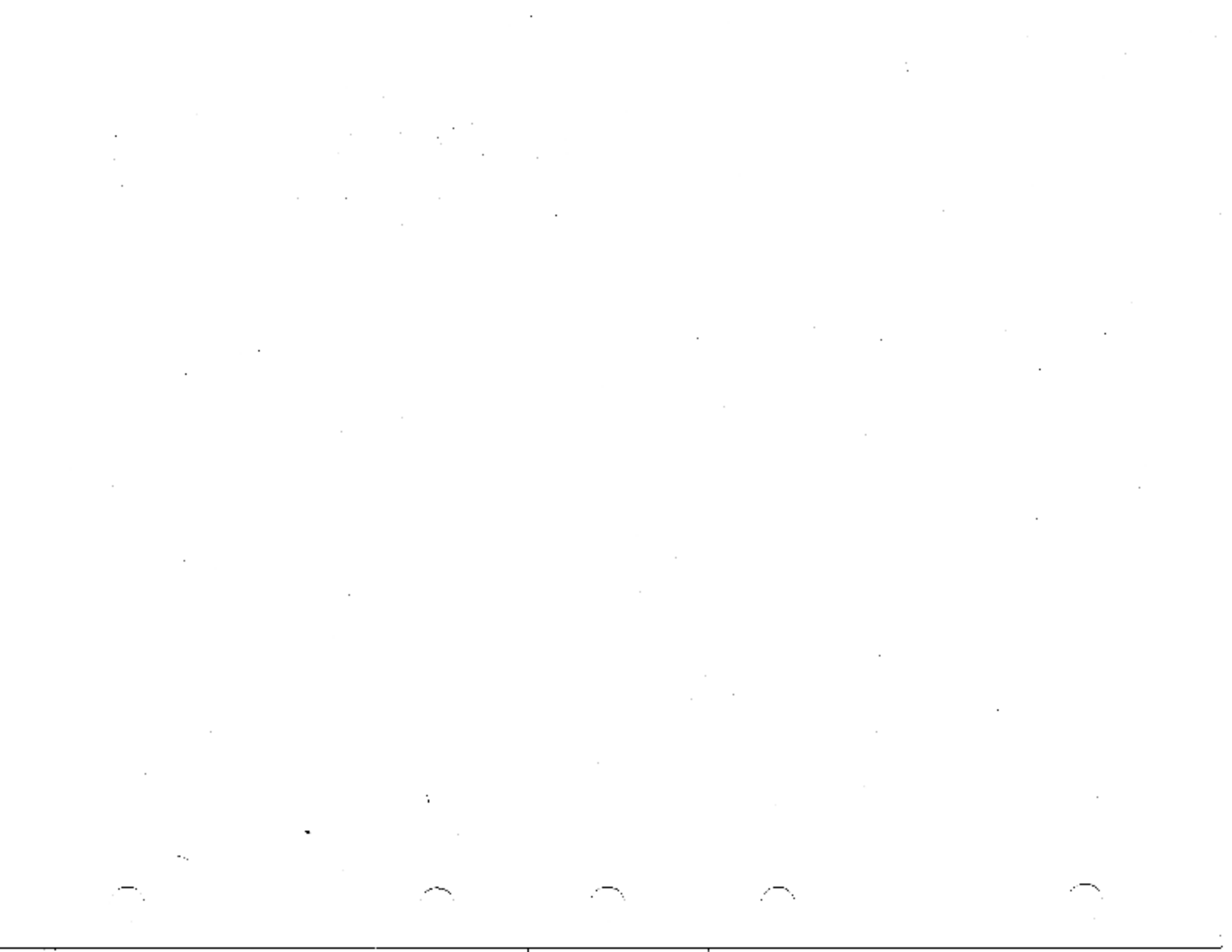
/DEBUG
: Specify /DEBUG to request the debugger at execution time if the image was not linked with the debugger. However, you cannot use /DEBUG if /NOTRACEBACK was specified when the image was linked.

/NODEBUG
: Specify /NODEBUG if you do not want the debugger at execution time for an image that was linked with the /DEBUG qualifier.

Examples

`RUN PROGA`	Executes PROGA.EXE.
`RUN PROGB.ABC`	Executes the image named PROGB.ABC.
`RUN/NODEBUG PROGA`	Executes PROGA.EXE without the debugger that may have been linked with it.

You can also use the RUN (Process) command to execute the image as a separate process. (See the **VAX/VMS Command Language User's Guide.**)



CHAPTER 3

NON-NUMERIC CHARACTER HANDLING

3.1 INTRODUCTION

COBOL programs hold their data in fields whose sizes are described in their source programs. These fields are thus "fixed" during compilation to remain the same size throughout the lifespan of the resulting object program.

The data descriptions of the fields in a COBOL program describe them as belonging to any of three data classes -- alphanumeric, alphabetic, or numeric class. Numeric class data items contain only numeric values, alphabetic class only A-Z and space, but alphanumeric class data items may contain values that are all alphabetic, all numeric, or a mixture of alphabetic bytes, numeric bytes, or, in fact, any character from the ASCII character set.

Further, these three classes are subdivided into five categories: alphabetic, numeric, numeric edited, alphanumeric edited, and alphanumeric. Every elementary item except for an index data item belongs to one of the classes and further to one of the categories. The class of a group item is treated at run time as alphanumeric regardless of the classes of subordinate elementary items.

For alphabetic and numeric (data items) class and category are synonymous.

An alphabetic field is a field declared to contain only alphabetic (A-Z and space) characters.

An alphanumeric class field that is declared to contain any ASCII character is called an alphanumeric category field.

If the data description of an alphanumeric class field specifies that certain editing operations will be performed on any value that is moved into it, that field is called an alphanumeric or numeric edited category field.

When reading the following sections of this chapter, this distinction between the class or category of a data item and the actual value that the item contains should always be kept in mind.

Sometimes the text refers to alphabetic, alphanumeric, and alphanumeric edited data items as non-numeric data items. This is to distinguish them from items that are specifically described as numeric items.

Regardless of the class of an item, it is usually possible to store a value in the item, at run time, that is "illegal". Thus, non-numeric ASCII characters can be placed into a field described as numeric class, and an alphabetic class field may be loaded with non-alphabetic characters.

To increase readability, the following sections occasionally omit the word "class" when describing an item; however, the reader should regard the descriptive word, numeric, alphabetic, or alphanumeric, as referring to the class of an item unless it applies specifically to the value in the item.

This chapter discusses non-numeric class data and the non-arithmetic, non-input-output operations that manipulate this type of data.

3.2 DATA ORGANIZATION

Usually, the data areas in a COBOL program are organized into group items with subordinate elementary items. A group item is a data item that is followed by one or more data items (elementary items) with higher valued level numbers. An elementary item has no higher valued subordinate level number.

All of the data areas used by COBOL programs (except for certain registers and switches) must be described in the Data Division of the source program. The compiler allocates memory space for these fields and fixes them in size at compilation time.

The following sub-sections (3.2.1 and 3.2.2) discuss, on a general level, how the compiler handles group and elementary data items.

3.2.1 Group Items

The size of a group item is the total size of the data area occupied by its subordinate elementary items. The compiler considers group items to be alphanumeric DISPLAY items. Thus, the software manipulates group items as if they had been described as PIC X() items, and ignores the structure of the data contained within them.

3.2.2 Elementary Items

The size of an elementary item is determined by the number of allowable symbols it contains that represent character positions. For example, consider figure 3-1.

```
01 TRANREC.
   03 FIELD-1 PIC X(7).
   03 FIELD-2 PIC S9(5)V99.
```

Figure 3-1
Field Sizes

Both fields consume seven bytes of memory; however, FIELD-1 contains seven alphanumeric bytes while FIELD-2 contains decimal digits and an operational sign. Although certain verbs handle these two classes of data differently, the data, in either case, occupies seven bytes of VAX-11 memory. COBOL operations on such fields are independent of the mapping of the field into VAX-11 memory words (16-bit words that hold two 8-bit bytes). Thus, a field may begin in the left or right-hand byte of a word with no effect on the function of any operations that may refer to that field.

In effect, the compiler sees memory as a continuous array of bytes, not words. This becomes particularly important when declaring a table with the OCCURS clause (see Chapter 5, Table Handling).

Records (a 01 level entry and all of its subordinate entries) and data items that have a level number of 77 and all literal values given in the Procedure Division automatically begin on even byte addresses.

I/O verbs require that records be aligned on word boundaries because the VAX-11 COBOL-74 file system reads and writes integral numbers of words.

Non-input-output verbs do not require alignment of the data. However, when two fields are aligned identically, the processing verb can sometimes increase its efficiency by processing them a word at a time rather that a byte at a time.

In all cases, automatic word alignment of literals, records, and/or 77 items increase the opportunity for more efficient processing.

3.3 SPECIAL CHARACTERS

COBOL allows the user to manipulate any of the 128 characters of the ASCII character set as alphanumeric data even though many of the characters are control characters, which usually control input/output devices. Generally, alphanumeric data manipulations are performed in a manner that attaches no "meaning" to an 8-bit byte. Thus, the user can move and compare these control characters in the same manner as alphabetic and numeric characters.

Although the object program can manipulate all ASCII characters, certain control characters cannot appear in non-numeric literals since the compiler uses them to delimit the source text. Further, the keyboards of the console and keypunch devices have no convenient input key for many of the special characters, thus making it difficult to place them into non-numeric literals.

Special characters may be placed into data fields of the object program by placing the binary value of the special character into a numeric COMP field and redefining that field as alphanumeric DISPLAY. Consider the following example of redefinition. (Keep in mind that the even byte of a word corresponds to the low-order bits of a binary word.)

```
01 LF-COMP PIC 999 COMP VALUE 10.
01 LF REDEFINES LF-COMP PIC X.
01 HT-COMP PIC 999 COMP VALUE 9.
01 TAB REDEFINES HT-COMP PIC X.
01 CR-COMP PIC 999 COMP VALUE 13.
01 CR REDEFINES CR-COMP PIC X.
```

Figure 3-2
Redefining Special Characters

The sample coding in Figure 3-2 introduces each character as a 1-word COMP item with a decimal value, then redefines it as a single byte. (The second byte of the redefinition need not be described at the 01 level, since redefinition at this level does not require identically sized fields.)

Use the Character Set table in Appendix B of the VAX-11 COBOL-74 Language Reference Manual to determine the decimal value for any ASCII character.

3.4 TESTING NON-NUMERIC FIELDS

3.4.1 Relation Tests

An IF statement that contains a relation condition (greater-than, less-than, equal-to, etc.) can compare the value in a non-numeric data item with another value and use the result to alter the flow of control in the program.

An IF statement with a relation condition compares two operands, either of which may be an identifier or a literal, except that both cannot be literals. If the relation exists between the two operands, the relation condition has a truth value of true.

Figure 3-3 illustrates the general format of a relation condition. (The relational characters ">," "<," and "=," although required, are not underlined to avoid confusion with other symbols such as greater-than-or-equal-to.)

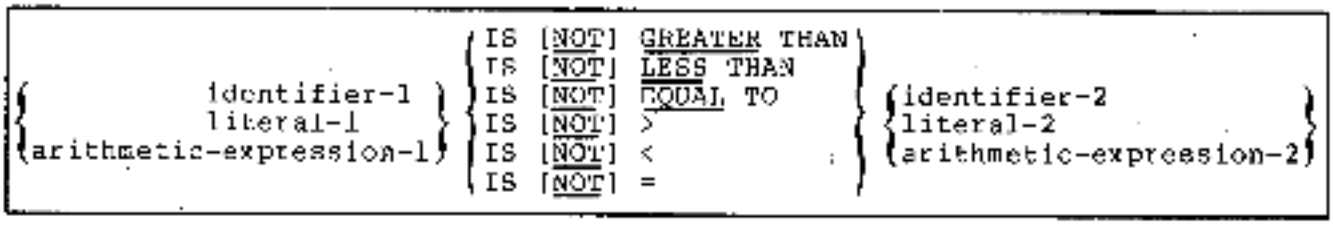


Figure 3-3
Relation Condition

When coding a relational operator, leave a space before and after each reserved word. When the reserved word NOT is present, the software considers it and the next key word or relational character to be one relational operator that defines the comparison. Figure 3-4 shows the meanings of the relational operators.

OPERATOR	MEANING
IS [NOT] GREATER THAN IS [NOT] >	The first operand is greater than (or not greater than) the second operand.
IS [NOT] LESS THAN IS [NOT] <	The first operand is less than (or not less than) the second operand.
IS [NOT] EQUAL TO IS [NOT] =	The first operand is equal to (or not equal to) the second operand.

Figure 3-4
The Meanings of the Relational Operators

3.4.1.1 Classes of Data - COBOL allows comparison of both numeric class operands and non-numeric class operands; however, it handles each class of data slightly differently. For example, it allows a comparison of two numeric operands regardless of the formats specified in their respective USAGE clauses, but requires that all other comparisons (including comparisons of any group items) be between operands with the same usage. It compares numeric class operands with respect to their algebraic values and non-numeric (or a numeric and a non-numeric) class operands with respect to a specified collating sequence.

If only one of the operands is numeric, it must be an integer data item or an integer literal and it must be DISPLAY usage; further, the manner in which the software handles numeric operands depends on the non-numeric operand. Consider the following two types of non-numeric operands:

1. If the non-numeric operand is an elementary item or a literal, the software treats the numeric operand as if it had been moved into an alphanumeric data item (which is the same size as the numeric operand) and then compared. This causes any operational sign, whether carried as a separate character or as an overpunch, to be stripped from the numeric item;

thus, it appears to be an unsigned quantity. In addition, if the picture-string of the numeric item contains trailing P characters indicating that there are assumed integer positions that are not actually present, these are filled with zero digits during the operation of stripping any sign that is present. Thus, an item with a picture-string of S9999PPP is moved to a temporary location where it is described with a picture-string of 9999999. If its value is 432J (-4321), the value in the temporary location will be 4321000. The numeric digits, stored as ASCII bytes, take part in the comparison.

2. If the non-numeric operand is a group item, the software treats the numeric operand as if it had been moved into a group item (which is the same size as the numeric operand) and then compared. This is equivalent to a "group move". The software ignores the description of the numeric field (except for length) and, therefore, includes any operational sign, whether carried as a separate character or as an overpunch, in its length. (Overpunched characters are never ASCII numeric digits, but characters in the range of from A through R, , or .) Thus, the sign and the digits, stored as ASCII bytes, take part in the comparison, and zeroes are not supplied for P characters in the picture-string.

The compiler will not accept a comparison between a non-integer numeric operand and a non-numeric operand, and any attempt to compare these two items will cause a diagnostic message at compile time.

3.4.1.2 The Comparison Operation - If the two operands are acceptable, the software compares them byte for byte starting at their left-hand end. It proceeds from left to right, comparing the characters in corresponding character positions until it either encounters a pair of unequal characters or reaches the right-hand end of the longer operand.

If the software encounters a pair of unequal characters, it considers their relative position in the collating sequence. The operand with the character that is positioned higher in the collating sequence is the greater operand.

If the operands have different lengths, the comparison proceeds as though the shorter operand were extended on the right by sufficient ASCII spaces (hex 20) to make them both the same length.

If all of the pairs of characters compare equally, the operands are equal.

3.4.2 Class Tests

An IF statement that contains a class condition (NUMERIC or ALPHABETIC) can test the value in a non-numeric data item (USAGE DISPLAY only) to determine if it contains numeric or alphabetic data and use the result to alter the flow of control in the program.

Figure 3-5 illustrates the general format of a class condition. If the data item consists entirely of the ASCII characters 0123456789 with or without the operational sign, the class condition would determine that it is NUMERIC. If the item consists entirely of the ASCII characters A through Z and space, the class condition would determine that it is ALPHABETIC.

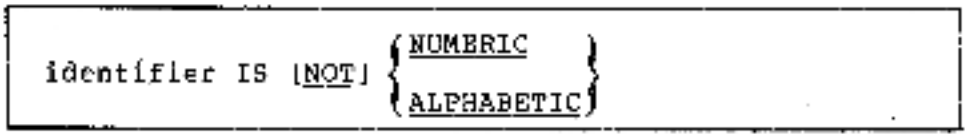


Figure 3-5
Class Condition, General Format

When the reserved word, NOT, is present, the software considers it and the next key word as one class condition that defines the class test to be executed; for example, NOT NUMERIC is a truth test for determining if an operand contains at least one non-numeric byte.

If the item being tested was described as a numeric data item, it may only be tested as NUMERIC or NOT NUMERIC. (For further information on using class conditions with numeric items, see Chapter 4.) The NUMERIC test cannot examine an item that was described either as alphabetic or as a group item containing elementary items whose data descriptions indicate the presence of operational signs.

3.5 DATA MOVEMENT

COBOL provides three statements (MOVE, STRING, and UNSTRING) that perform most of the data movement operations required by business-oriented programs. The MOVE statement simply moves data from one field to another. The STRING statement concatenates a series of sending fields into a single receiving field. The UNSTRING statement disperses a single sending field into multiple receiving fields. Each has its uses and its limitations. This section discusses data movement situations which take advantage of the versatility of these statements.

The MOVE statement handles the majority of data movement operations on character strings. However, the MOVE statement has limitations in its ability to handle multiple fields; for example, it cannot, by itself, concatenate a series of sending fields into a single receiving field or disperse a single sending field into several receiving fields.

Two MOVE statements will, however, bring the contents of two fields together into a third (receiving) field if the receiving field has been "subdivided" with subordinate elementary items that match the two sending fields in size. If other fields are to be concatenated into the third field and they differ in size from the first two fields, then the receiving field will require additional subdivisions (through redefinition).

Another method of concatenation with the MOVE statement is to subdivide the receiving field into single character fields, creating a "table" of a single character field that occurs as many times as there are characters in the receiving field, and execute a data movement loop which moves each sending field, a character at a time, using a subscript that moves continuously across the receiving field.

Two MOVE statements can also be used to disperse the contents of one sending field to several receiving fields. The first MOVE statement can move the left-most end of the sending field to a receiving field; then the second MOVE statement can move the right-most end of the sending field to another receiving field. (The second receiving field must first be described with the JUSTIFIED clause.) Characters from the middle of the sending field cannot easily be moved to any receiving field without extensive redefinitions of the sending field or a character-by-character movement loop (as with concatenation).

The concatenation and dispersion limitations of the MOVE statement are handled quite easily by the STRING and UNSTRING statements. The following sections (3.6, 3.7, and 3.8) discuss these three statements in detail.

3.6 THE MOVE STATEMENT

The MOVE statement moves the contents of one field into another. The following illustration shows the two formats of the MOVE statement.

Format 1

```
MOVE FIELD1 TO FIELD2
```

Format 2

```
MOVE CORRESPONDING FIELD1 TO FIELD2
```

NOTE

Format 2 is discussed in Section 3.6.6.

FIELD1 is the name of the sending field and FIELD2 is the name of the receiving field. The statement causes the software to move the contents of FIELD1 into FIELD2. The two fields need not be the same size, class, or usage; and they may be either group or elementary items.

If the two fields are not the same length, the software will align them on one end or the other -- and will truncate or pad (with spaces) the other end. The movement of group items and non-numeric elementary items is discussed below.

A point to remember when using the MOVE statement is that it will alter the contents of every character position in the receiving field.

3.6.1 Group Moves

If either the sending or receiving field is a group item, the software considers the move to be a group move. It treats both the sending and receiving fields in a group move as if they were alphanumeric class fields. If the sending field is a group item and the receiving field is an elementary item, the software ignores the receiving field description (except for the size description, in bytes, and any JUSTIFIED clause); therefore, the software conducts no conversion or editing on the receiving field.

3.6.2 Elementary Moves

If both fields of a MOVE statement are elementary items, their data description clauses control their data movement. (If the receiving field was described as numeric or numeric edited, the rules for numeric moves -- see Chapter 4, Numeric Character Handling -- control the data movement.)

The following table shows the legal (and illegal) non-numeric elementary moves.

Table 3-1
Legal Non-Numeric Elementary Moves

SENDING FIELD CATEGORY	RECEIVING FIELD CATEGORY	
	ALPHABETIC	ALPHANUMERIC ALPHANUMERIC EDITED
ALPHABETIC	Legal	Legal
ALPHANUMERIC	Legal	Legal
ALPHANUMERIC EDITED	Legal	Legal
NUMERIC INTEGER (DISPLAY ONLY)	Illegal	Legal
NUMERIC EDITED	Illegal	Legal

In all of the legal moves shown above, the software treats the sending field as though it had been described as PIC X(). If the sending field description contains a JUSTIFIED clause, the clause will have no effect on the move. If the sending field picture-string contains editing characters, the software uses them only to determine the field's size.

Numeric class data must be in DISPLAY (byte) format and must be an integer.

If the description of the numeric data item indicates the presence of an operational sign (either as a character or an overpunch) or if there are P characters in the picture-string of the numeric data item, the software first moves the item to a temporary location. During this move, it removes the sign and fills out any P character positions with zero digits. It then uses the temporary value (which may be shorter than the original if a separate sign were removed, or longer if P character positions were filled in with zeroes) as the sending field as if it had been described as PIC X(), that is, as if its category were alphanumeric.

If the sending item is an unsigned numeric class field with no P characters in its picture-string, the software does not move the item to a temporary location.

A numeric integer data item sending field has no effect on the justification of the receiving field. If the numeric sending field is shorter than the receiving field, the software fills the receiving field with spaces.

In legal, non-numeric elementary moves, the receiving field actually controls the movement of data. All of the following items, in the receiving field, affect the move: (1) the size, (2) the presence of editing characters in its description, and (3) the presence of the JUSTIFIED RIGHT clause in its description. The JUSTIFIED clause and editing characters are mutually exclusive; therefore, the two classes are discussed separately below.

When a field that contains no editing characters or JUSTIFIED clause in its description is used as the receiving field of a non-numeric elementary MOVE statement, the statement moves the characters by starting at the left-hand end of the fields and scanning across, character-by-character to the right. If the sending item is shorter than the receiving item, the software fills the remaining character positions with spaces.

3.6.2.1 Edited Moves - Alphabetic or alphanumeric fields may contain editing characters. Consider the following insertion editing characters. Alphabetic fields will accept only the B character; however, alphanumeric fields will accept all three characters.

B -- blank insertion position
0 -- zero insertion position
/ -- slash insertion position.

When a field that contains an insertion editing character in its picture-string is used as the receiving field of a non-numeric elementary MOVE statement, each receiving character position that corresponds to an editing character receives the insertion byte value. Figure 3-6 illustrates the use of such symbols with the statement, MOVE FIELD1 TO FIELD2. (Assume that FIELD1 was described as PIC X(7).)

	FIELD2	
FIELD1	PICTURE-STRING	CONTENTS AFTER MOVE
070476	XX/99/XX	07/04/76
04JUL76	99BAAAB99	04 JUL 76
2351212	XXXBXXXX/XX/	235 1212/ /
123456	0XB0XB0XB0X	01 02 03 04

Figure 3-6
Data Movement with Editing Symbols

Data movement always begins at the left end of the sending field, and moves only to the byte positions described as A, 9, or X in the receiving field picture-string. When the sending field is exhausted, the software supplies space characters to fill any remaining character positions (not insertion positions) in the receiving field. If the receiving field becomes exhausted before the last character is moved from the sending field, the software ignores the remaining sending field characters.

3.6.2.2 Justified Moves - A JUSTIFIED RIGHT clause in the data description of the receiving field causes the software to reverse its usual data movement conventions. (It starts with the right-hand characters of both fields and proceeds from right to left.) If the sending field is shorter than the receiving field, the software fills the remaining left-hand character positions with spaces. Figure 3-7 illustrates various data description situations for the statement, MOVE FIELD1 TO FIELD2, with no editing.

FIELD1		FIELD2	
PICTURE-STRING	CONTENTS	PICTURE-STRING (AND JUST CLAUSE)	CONTENTS AFTER MOVE
XXX	ABC	XX XXXXX XX JUST XXXXX JUST	AB ABC BC ABC

Figure 3-7
Data Movement with No Editing

3.6.3 Multiple Receiving Fields

If a MOVE statement is written with more than one receiving field, it moves the same sending field value to each of the receiving fields. It has essentially the same effect as a series of separate MOVE statements that all have the same sending field. (For information on subscripted fields, see section 3.6.4.)

The receiving fields need have no relationship to each other. The software checks the legality of each one independently, and performs an independent move operation on each one.

Multiple receiving fields on MOVE statements provide a convenient way to set many fields equal to the same value, such as during initialization code at the beginning of a section of processing. For example:

```
MOVE SPACES TO LIST-LINE, EXCEPTION-LINE, NAME-FLD.

MOVE ZEROES TO EOL-FLAG, EXCEPT-FLAG, NAME-FLAG.

MOVE 1 TO COUNT-1, CHAR-PTR, CURSOR.
```

3.6.4 Subscripted Moves

Any field of a MOVE statement may be subscripted and the referenced field may also be used to subscript another name in the same statement.

When more than one receiving field is named in the same MOVE statement, the order in which the software evaluates the subscripts affects the results of the move. Consider the following two situations:

```
Situation 1  MOVE FIELD1(FIELD2) TO FIELD2 FIELD3.

Situation 2  MOVE FIELD1 TO FIELD2 FIELD3(FIELD2).
```

Figure 3-8
Subscripted MOVE Statements

In situation 1, the software evaluates FIELD1(FIELD2) only once, before it moves any data to the receiving fields. In effect it is as if the statement were replaced with the following statements:

```
MOVE FIELD1(FIELD2) TO TEMP.

MOVE TEMP TO FIELD2.

MOVE TEMP TO FIELD3.
```

In situation 2, the software evaluates FIELD3(FIELD2) immediately before moving the data into it (but after moving the data from FIELD1 to FIELD2). Thus, it uses the newly stored value of FIELD2 as the subscript value. In effect, it is as if the statement were replaced with the following statements:

```
MOVE FIELD1 TO FIELD2.

MOVE FIELD1 TO FIELD3(FIELD2).
```

3.6.5 Common Errors, MOVE Statement

A most important thing to remember when writing MOVE statements is that the compiler considers any MOVE statement that contains a group item to be a group move. It is easy to forget this fact when moving a group item to an elementary item, and the elementary item contains editing characters, or a numeric integer. These attributes of the receiving field (which would determine the action of an elementary move) have no effect on the action of a group move.

3.6.6 Format 2 - MOVE CORRESPONDING

Format 2 of the MOVE statement allows the programmer to move multiple elementary items from one group item to another, by using a single MOVE statement. When the corresponding phrase is used, selected elementary items in the sending field are moved to those elementary items in the receiving field whose data-names are identical. For example:

```
01 A-GROUP.                    01 B-GROUP.

   02 FIELD1.                     02 FIELD2.

      03 A PIC X.                    03 A PIC X.

      03 B PIC 9.                    03 C PIC XX.

      03 C PIC XX.                   03 E PIC XXX.

      03 D PIC 99.

      03 E PIC XXX.

MOVE CORRESPONDING A-GROUP TO B-GROUP

                  OR

MOVE CORRESPONDING FIELD1 TO FIELD2
```

The preceding examplee are equlvalent to the following series of MOVE statements:

```
MOVE A OF FIELD1 TO A OF FIELD2

MOVE C OF FIELD1 TO C OF FIELD2

MOVE E OF FIELD1 TO E OF FIELD2
```

3.7 THE STRING STATEMENT

The STRING statement concatenates the contents of two or more sending fields into a single field.

The statement has many forms; the simplest is equivalent, in function, to a non-numeric MOVE statement. Consider the following illustration; if the two fields are the same size, or if the sending field (FIELD1) is larger, the statement is equivalent to the statement, MOVE FIELD1 TO FIELD2.

```
STRING  FIELD1 DELIMITED BY SIZE INTO FIELD2.
```

Figure 3-9
Sample STRING Statement

If the sending field is shorter than the receiving field, an important difference between the STRING and MOVE statements emerges: the software does not fill the receiving field with spaces. Thus, the STRING statement may leave some portion of the receiving field unchanged.

Additionally, the receiving field must be an elementary alphanumeric field with no JUSTIFIED clause or editing characters in its description. Thus, the data movement of the STRING statement always fills the receiving field from left-to-right with no editing insertions.

3.7.1 Multiple Sending Fields

An important characteristic of the STRING statement is its ability to concatenate a series of sending fields into one receiving field. Consider the following example of the STRING statement:

```
STRING FIELD1A FIELD1B FIELD1C DELIMITED BY SIZE
                    INTO FIELD2.
```

Figure 3-10
Concatenation with the STRING Statement

In this example STRING statement, FIELD1A, FIELD1B, and FIELD1C are all sending fields. The software moves them to the receiving field (FIELD2) in the order in which they appear in the statement, from left to right, resulting in the concatenation of their values.

If FIELD2 is not large enough to hold all three items, the operation stops when it is full. If this occurs while moving one of the sending fields, the software ignores the remaining characters of that field and any other sending fields not yet processed. For example, if FIELD2 became full while receiving FIELD1B, the software would ignore the rest of FIELD1B and all of FIELD1C.

If the sending fields do not fill the receiving field, the operation stops with the movement of the last character of the last sending item (FIELD1C in Figure 3-10). The software does not alter the contents nor space-fill the remaining character positions of the receiving field.

The sending fields may be non-numeric literals and figurative constants (except for ALL literal). For example, the following statement sets up an address label with the literal period and space between the STATE and ZIP fields:

```
STRING CITY SPACE STATE ".  " ZIP
        DELIMITED BY SIZE INTO ADDRESS-LINE.
```

Figure 3-11
Literals as Sending Fields

Sending fields may also be subscripted. For example, the following statement uses subscripts to concatenate the elements of a table (A-TABLE) into a single field (A-FOUR). (I, of course, must be a subscript or an index-name.)

```
STRING A-TABLE(I) A-TABLE(I+1) A-TABLE(I+2) A-TABLE(I+3)
        DELIMITED BY SIZE INTO A-FOUR.
```

Figure 3-12
Indexed Sending Fields

3.7.2 The POINTER Phrase

Although the STRING statement normally starts at the left-hand end of the receiving field, with the POINTER phrase it is possible to start it scanning at another point within the field. (The scanning, however, remains left-to-right.)

```
MOVE 5 TO P.
STRING FIELD1A FIELD1B DELIMITED BY SIZE
        INTO FIELD2 WITH POINTER P.
```

Figure 3-13
Sample POINTER Phrase

When the POINTER phrase is used, the value of P determines the starting character position in the receiving field. In Figure 3-13, the 5 in P causes the software to move the first character of FIELD1A into character position 5 of FIELD2 (the left-most character position of the receiving field is character position 1) and leave positions 1 through 4 unchanged.

When the STRING operation is complete, the software leaves P pointing to one character position beyond the last character replaced in the receiving field. If FIELD1A and FIELD1B in Figure 3-13 are both four characters long, P will contain a value of 13 (5+4+4) when the operation is complete (assuming that FIELD2 is at least 12 characters long).

3.7.3 The DELIMITED BY Phrase

Although the sending fields of the STRING statement are fixed in size at compile time, they frequently contain variable-length items that are padded with spaces. For example, a 20-character city field may contain only the word MAYNARD and 13 spaces. A valuable feature of the STRING statement is that it may be used to move only the useful data from the left-hand end of the sending field. The DELIMITED BY phrase, written with a data-name or literal, instead of the word SIZE, performs this operation. (The delimiter may be a literal, a data item, a figurative constant, or the word SIZE. It may not be ALL literal since ALL literal has an indefinite length. When the phrase contains the word SIZE, the software moves each sending field, in total, until it either exhausts the sending field, or fills the receiving field.)

Consider the following example:

```
STRING CITY SPACE STATE ",  " ZIP
        DELIMITED BY SIZE INTO ADDRESS-LINE.
```

Figure 3-14
Delimiting with the Word SIZE

If CITY is a 20-character field, the result of the STRING operation shown in Figure 3-14 might look like the following:

```
AYER_____MA,  01432
     \_____16 spaces
```

A far more attractive printout can be produced by having the STRING operation produce the following:

```
AYER, MA.  01432
```

To accomplish this, use the figurative constant SPACE as a delimiter on the sending field; thus,

```
MOVE 1 TO P.
STRING CITY DELIMITED BY SPACE
          INTO ADDRESS-LINE WITH POINTER P.
STRING ", " STATE ". " ZIP
          DELIMITED BY SIZE
          INTO ADDRESS-LINE WITH POINTER P.
```

Figure 3-15
SPACE as a Delimiter

This sample coding uses the pointer's characteristic of pointing to one character position beyond the last character replaced in the receiving field to enable the second STRING statement to begin at a position one character past where the first STRING statement stopped. (The first STRING statement moves data characters until it encounters a space character -- a match of the delimiter SPACE. The second STRING statement adds the literal, the 2-character STATE field, another literal, and the 5-character ZIP field.)

The delimiter can be varied for each field within a single STRING statement by repeating the DELIMITED BY phrase after the sending field names to which it applies. Thus, the following shorter statement has the same effect as the preceding example. (Placing the operands on separate source lines, as shown in this example, has no effect on the operation of the statement, but improves program readability and simplifies debugging.)

```
STRING CITY DELIMITED BY SPACE
            ", " STATE ". "
            ZIP DELIMITED BY SIZE
                INTO ADDRESS-LINE.
```

Figure 3-16
Repeating the DELIMITED BY Phrase

The sample STRING statement in Figure 3-16 cannot handle 2-word city names, such as New York, since the software would consider the space between the two words as a match for the delimiter SPACE. A longer delimiter, such as two or three spaces (non-numeric literal), can solve this problem. Only when a sequence of characters matches the delimiter will the movement stop for that data item.

With a 2-byte delimiter, the same statement can be rewritten in a simpler form:

```
STRING CITY ", " STATE ", " ZIP
            DELIMITED BY "  " INTO ADDRESS-LINE.
```

Figure 3-17
Delimiting with More Than One Space Character

Since only the CITY field may contain two consecutive spaces (the entire STATE field is only two bytes long), the delimiter's search of the other fields will always be unsuccessful and the effect is the same as moving the full field (delimiting by SIZE).

Data movement under control of a data-name or literal is generally slower in execution speed than movement delimited by SIZE.

The example in Figure 3-17 illustrates a frequent source of error in the use of STRING statements to concatenate fields. The remainder of the receiving field is not space-filled as with a MOVE statement. If ADDRESS-LINE is to be printed on a mailing label, for example, the STRING statement should be preceded by the statement, MOVE SPACES TO ADDRESS-LINE. This guarantees a space fill to the right of the concatenated result. Alternatively, the last field concatenated by the STRING statement can be a field previously set to SPACES. (This sending field must be moved under control of a delimiter other than SPACE, of course.)

3.7.4 The OVERFLOW Phrase

When the SIZE option of the DELIMITED BY phrase controls the STRING operation and the pointer value is either known or the POINTER phrase is not used, the programmer can tell, by simple addition, if the receiving field is large enough to hold the sending fields. However, if the DELIMITED BY phrase contains a literal or an identifier, or if the pointer value is not predictable, it may be difficult to tell whether the size of the receiving field is adequate, and an overflow may occur.

The ON OVERFLOW phrase at the end of the STRING statement tests for an overflow condition:

```
STRING FIELD1A FIELD1B DELIMITED BY "C"
                INTO FIELD2 WITH POINTER PNTR
                ON OVERFLOW GO TO PN57.
```

Figure 3-18
The ON OVERFLOW Phrase

Overflow occurs when the receiving field is full and the software is either about to move a character from a sending field or is considering a new sending field. Overflow may also occur if, during the initialization of the statement, the pointer contains a value that is either less than 1 or greater than the length of the receiving field. In this case, the software moves no data to the receiving field and terminates the operation immediately.

The ON OVERFLOW phrase cannot distinguish the overflow caused by a bad initial value in pointer PNTR from the overflow caused by a receiving field that is too short. Only a separate test, preceding the STRING statement, can distinguish between the two.

The following examples illustrate the overflow condition:

```
     DATA DIVISION.
          ...
          01 FIELD1A PIC XXX VALUE "ABC".
          01 FIELD2 PIC XXXX.


       PROCEDURE DIVISION.
          ...
    1.    STRING FIELD1A QUOTE DELIMITED BY SIZE INTO FIELD2.
    2.    STRING FIELD1A FIELD1A DELIMITED BY SIZE INTO FIELD2.
    3.    STRING FIELD1A FIELD1A DELIMITED BY "C" INTO FIELD2.
    4.    STRING FIELD1A FIELD1A FIELD1A FIELD1A
               DELIMITED BY "B" INTO FIELD2.
    5.    STRING FIELD1A FIELD1A "C" DELIMITED BY "C"
               INTO FIELD2.
    6.    MOVE 2 TO P.
          STRING FIELD1A "AC" DELIMITED BY "C"
               INTO FIELD2 WITH POINTER P.
```

Figure 3-19
Various STRING Statements
Illustrating the Overflow Condition

The results of executing the numbered statements follow:

Table 3-2
Results of the
Preceding Sample Statements

Value of FIELD2 after the STRING operation	Overflow?
1. ABC"	No
2. ABCA	Yes
3. ABAB	No
4. AAAA	No
5. ABAB	Yes
6. AABA	No

3.7.5 Subscripted Fields in STRING Statements

All data-names used in the STRING statement may be subscripted, and the pointer value may be used as a subscript.

Since the pointer value might be used as a subscript on one or more of the fields in the statement, it is important to understand the order in which the software evaluates the subscripts and exactly when it updates the pointer. (The use of the pointer as a subscript is not specified by ANS-74 COBOL. Before using it, read the note at the end of this subsection.)

The software updates the pointer after it moves the last character out of each sending field. Consider the following sample coding:

```
MOVE 1 TO P.
STRING "ABC"
       SPACE
       "DEF" DELIMITED BY SIZE
       INTO R WITH POINTER P.
```

Figure 3-20
STRING Statement with Pointer

During the movement of "ABC" into the receiving field (R), the pointer value remains at 1. After the move, the software increases the pointer value by 3 (the size of the sending field literal "ABC") and it takes on the value 4. The software then moves the figurative constant SPACE and increases the pointer value by 1 and it takes on the value 5. "DEF" is then moved and, on completion of the move, the software increases the pointer to its final value for this operation, 8.

Now, consider the updating characteristics of the pointer when applied to subscripting:

```
MOVE 1 TO P.
STRING CHAR(P)
       CHAR(P)
       CHAR(P)
       CHAR(P) DELIMITED BY SIZE
       INTO R WITH POINTER P.
```

Figure 3-21
Subscripting with the Pointer

If CHAR is a 1-character field in a table, the pointer increases by one after each field has been moved and the software will move them into R as if they had been subscripted as CHAR(1), CHAR(2), CHAR(3), and CHAR(4). If CHAR is a 2-character field, the pointer increases by two after each field has been moved and the fields will move into R as if they had been subscripted as CHAR(1), CHAR(3), CHAR(5), and CHAR(7).

Thus, the software evaluates the subscript of a sending item once, immediately before it considers the item as a sending item.

The software evaluates the subscript of a receiving item only once, at the start of the STRING operation. Therefore, if the pointer is used as a subscript on the receiving field, changes occurring to the pointer during the execution of the STRING statement will not alter the choice of which receiving string is altered.

Even the delimiter field can be subscripted, and it too can be subscripted with the pointer. The software re-evaluates the delimiter subscript once for each sending field, immediately before it compares the delimiter to the field. Thus, by subscripting it with the pointer value, the delimiter can be changed for each sending field. This has the peculiar effect of choosing the next sending field's delimiter based on the position, in the receiving field, into which its first character will fall. For example, consider the following sample coding:

```
01 DTABLE.
   03 D PIC X OCCURS 7 TIMES.


   MOVE 1  TO P.
   STRING  "ABC"
           "ABC"
           "ABC" DELIMITED BY D(P)
           INTO R WITH POINTER P.
```

Figure 3-22
Subscripting the Delimiter

The following table shows the value that will arrive in the receiving field (R) from the three "ABC" literals if DTABLE contains the values shown in the left-hand column:

Table 3-3
Results of the
Preceding Sample Statements

DTABLE Value	R Value
ABCDEFG	(Unchanged)
BCDEFGH	AABABC
CDEFGH1	ABABCABC
CCCCCCC	ABABAB

NOTE

The rules in this section, concerning subscripts in the STRING statement, are rules that are not specified by 1974 American National Standard COBOL. Dependence on these rules, particularly those involving the use of the pointer field as a subscript, may produce programs that will not perform the same way on other COBOL compilers.

If the pointer field is not used as a subscript on any of the fields in the statement, the point at which the software evaluates the subscripts is immaterial to the execution of the statement. Thus, by avoiding the use of the pointer as a subscript, uniform results can be expected from all COBOL compilers that adhere to 1974 ANS COBOL.

3.7.6 Common Errors, STRING Statement

The most common errors made when writing STRING statements are:

- using the word "TO" instead of "INTO"
- forgetting to write "DELIMITED BY SIZE";
- forgetting to initialize the pointer;
- initializing the pointer to 0 instead of 1;
- forgetting to provide for space fill of the receiving field when it is desirable.

3.8 THE UNSTRING STATEMENT

The UNSTRING statement disperses the contents of a single sending field into multiple receiving fields.

The statement has many forms; the simplest is equivalent in function to a non-numeric MOVE statement. Consider the following illustration; the sample statement is equivalent to MOVE FIELD1 TO FIELD2, regardless of the relative sizes of the two fields.

```
UNSTRING FIELD1 INTO FIELD2.
```

Figure 3-23
Sample UNSTRING Statement

The sending field (FIELD1) may be either a group item or an alphanumeric, or alphanumeric edited elementary item. The receiving field (FIELD2) may be alphabetic, alphanumeric, or numeric, but it cannot specify any type of editing.

If the receiving field is numeric, it must be DISPLAY usage. The picture-string of a numeric receiving field may contain any of the legal numeric description characters except for P and, of course, the editing characters. The UNSTRING statement moves the sending field to numeric receiving fields as if the sending field had been described as an unsigned integer; further, it automatically truncates or zero fills as required.

If the receiving field is not numeric, the software follows the rules for elementary non-numeric MOVE statements. It left-justifies the data in the receiving field, truncating or space-filling as required. (If the data-description of the receiving field contains a JUSTIFIED clause, the software right-justifies the data, truncating or space-filling to the left as required.)

3.8.1 Multiple Receiving Fields

An important characteristic of the UNSTRING statement is its ability to disperse one sending field into several receiving fields. Consider the following example of the UNSTRING statement written with multiple receiving fields:

```
UNSTRING FIELD1 INTO
        FIELD2A FIELD2B FIELD2C.
```

Figure 3-24
Multiple Receiving Fields

In this sample statement, FIELD1 is the sending field. The software performs the UNSTRING operation by scanning across FIELD1 from left to right. When the number of characters scanned is equal to the number of characters in the receiving field, the software moves the scanned characters into the receiving field and begins scanning the next group of characters for the next receiving field.

Assume that each of the receiving fields in the preceding illustration (FIELD2A, FIELD2B, and FIELD2C) is five characters long, and that FIELD1 is 15 characters long. The size of FIELD2A determines the number of characters for the first move. The software scans across FIELD1 until the number of characters scanned equals the size of FIELD2A (5). It then moves those first five characters to FIELD2A, and sets the scanner to the next (sixth) character position in FIELD1. The size of FIELD2B determines the size of the next move. The software begins this move by scanning across FIELD1 from character position six, until the number of scanned characters equals the size of FIELD2B (5).

The software then moves the sixth through the tenth characters to FIELD2B, and sets the scanner to the next (eleventh) character position in FIELD1. FIELD2C determines the size of the last move (for this example) and causes characters 11 through 15 of FIELD1 to be moved into FIELD2C, thus terminating this UNSTRING operation.

Each data movement acts as an individual MOVE statement, the sending field of which is an alphanumeric field equal in size to the receiving field. If the receiving field is numeric, the move operation will convert the data to the numeric form.. For example, consider what would happen if the fields under discussion had the data descriptions and were manipulating the values shown in the following table:

Table 3-4
Values Moved Into the Receiving Fields
Based on the Value in the Sending Field

FIELD1 PIC X(15). VALUE IS:	FIELD2A PIC X(5)	FIELD2B PIC S9(5) LEADING SEPARATE	FIELD2C PIC S999V99
ABCDE1234512345	ABCDE	+12345	3450
XXXXX0000100123	XXXXX	+00001	1230

FIELD2A is an alphanumeric field and, therefore, the software simply conducts an elementary non-numeric move with the first five characters.

FIELD2B, however, has a leading separate sign that is not included in its size. Thus, the software moves only five numeric characters and generates a positive sign in the separate sign position.

FIELD2C has an implied decimal point with two character positions to the right of it, plus an overpunched sign on the low-order digit. The sending field should supply five numeric digits; but, since the sending field is alphanumeric, the software treats it as an unsigned integer; it truncates the two high-order digits and supplies two zero digits for the decimal positions. Further, it supplies a positive overpunch sign, making the low-order digit a +0 (or the ASCII character, {). (There is no simple way to have UNSTRING recognize a sign character or a decimal point in the sending field.)

If the sending field is shorter than the sum of the sizes of the receiving fields, the software ignores the remaining receiving fields. If it reaches the end of the sending field before it reaches the end of one of the receiving fields, the software moves the scanned characters into that receiving field. It left-justifies and fills the remaining character positions with spaces for alphanumeric data, or decimal point aligns and zero fills the remaining character positions for numeric data.

Consider the following examples of a sending field that is too short. (The statement is UNSTRING FIELD1 INTO FIELD2A FIELD2B. FIELD2A is a 3-character alphanumeric field, and receives the first three characters of FIELD1 (ABC) in every operation. FIELD2B, however, runs out of characters every time before filling. Since FIELD2A always contains the characters ABC, it is not shown.)

Table 3-5
Handling a Sending Field that is Too Short

FIELD1 PIC X(6) VALUE IS:	FIELD2B PICTURE IS:	FIELD2B Value after UNSTRING Operation
ABCDEF	XXXXX	DEF
ABC246	S99999	0024F
	S9V999	600
	S9999	+0246
	LEADING SEPARATE	

3.8.2 The DELIMITED BY Phrase

The size of the data to be moved can be controlled by a delimiter, rather than by the size of the receiving field. The DELIMITED BY phrase supplies the delimiter characters.

UNSTRING delimiters are quite flexible; they can be literals, figurative constants (including ALL literal), or identifiers (identifiers may even be subscripted data-names). This sub-section discusses the use of these three types of delimiters. Subsequent sections cover multiple delimiters, the COUNT phrase, and the DELIMITER phrase. Subscripting delimiters is discussed at the end of this section under Subscripted Fields in UNSTRING Statements.

Consider the following sample UNSTRING statement; it uses the figurative constant, SPACE, as a delimiter:

```
UNSTRING FIELD1 DELIMITED BY SPACE INTO FIELD2.
```

Figure 3-25
Delimiting with a Space Character

In this example, the software scans the sending field (FIELD1), searching for a space character. If it encounters a space, it moves all of the scanned (non-space) characters that precede that space to the receiving field (FIELD2). If it finds no space character, it moves the entire sending field. When it has determined the size of the sending field, the software moves the contents of that field following the rules for the MOVE Statement, truncating or zero filling as required.

The following table shows the results of an UNSTRING operation that delimits with a literal asterisk (UNSTRING FIELD1 DELIMITED BY "*" INTO FIELD2).

Table 3-6
Results of Delimiting with an Asterisk

FIELD1 PIC X(6) VALUE IS:	FIELD2 PICTURE IS:	FIELD2 VALUE AFTER UNSTRING
ABCDEF	XXX	ABC
	X(7)	ABCDEF
	XXX JUSTIFIED	DEF
******	XXX	
*ABCDE	XXX	
A*****	XXX JUSTIFIED	A
246***	S9999	0246
12345*	S9999 SEPARATE TRAILING	2345+
2468**	S999V9 SEPARATE LEADING	+4680
*246**	9999	0000

If the delimiter matches the first character in the sending field, the software considers the size of the sending field to be zero. The movement operation still takes place, however, and fills the receiving field with spaces or zeroes depending on its class.

A delimiter may also be applied to an UNSTRING statement that has multiple receiving fields:

```
UNSTRING FIELD1 DELIMITED BY SPACE
        INTO FIELD2A FIELD2B.
```

Figure 3-26
Delimiting with Multiple Receiving Fields

The sample instruction in Figure 3-26 causes the software to scan FIELD1 searching for a character that matches the delimiter. If it finds a match, it moves the scanned characters to FIELD2A and sets the scanner to the next character position to the right of the character that matched. It then resumes scanning FIELD1 for a character that matches the delimiter.

If the software finds a match, it moves all of the characters that lie between the character that first matched the delimiter and the character that matched on the second scan, and sets the scanner to the next character position to the right of the character that matched. (The DELIMITED BY phrase could handle additional receiving fields in the same manner as it handled FIELD2B.)

The following table shows the results of an UNSTRING operation that applies a delimiter to multiple receiving fields (UNSTRING FIELD1 DELIMITED BY "*" INTO FIELD2A FIELD2B).

Table 3-7
Results of Delimiting
Multiple Receiving Fields

FIELD1 PIC X(8) VALUE IS:	VALUES AFTER UNSTRING OPERATION	
	FIELD2A PIC X(3)	FIELD2B PIC X(3)
ABC*DEF*	ABC	DEF
ABCDE*FG	ABC	FG
A*B*****	A	B
*AB*CD**		AB
**ABCDEF		
A*BCDEFG	A	BCD
ABC**DEF	ABC	
A******B	A	

The last two examples illustrate the limitations of a single character delimiter. Accordingly, the delimiter may be longer than one character and it may be preceded by the word ALL.

The following table shows the results of an UNSTRING operation that uses a 2-character delimiter (UNSTRING FIELD1 DELIMITED BY "**" INTO FIELD2A FIELD2B):

Table 3-8
Results of Delimiting
with Two Asterisks

FIELD1 PIC X(8) VALUE IS:	VALUES AFTER UNSTRING OPERATION	
	FIELD2A PIC XXX	FIELD2B PIC XXX JUSTIFIED
ABC**DEF	ABC	DEF
A*B*C*D*	A*B	
AB***C*D	AB	C*D
AB**C*D*	AB	*D*
AB**CD**	AB	CD
AB***CD*	AB	CD*
AB*****CD	AB	

Unlike the STRING statement, the UNSTRING statement accepts the ALL literal as a delimiter. When the word ALL precedes the delimiter, the action of the UNSTRING statement remains essentially the same as with one delimiter until the scanning operation finds a match. At this point, the software scans farther, looking for additional consecutive strings of characters that also match the delimiter item. It considers the "ALL delimiter" to be one, two, three, or more adjacent repetitions of the delimiter item.

The following table illustrates the results of an UNSTRING operation that uses an ALL delimiter (UNSTRING FIELD1 DELIMITED BY ALL "*" INTO FIELD2A FIELD2B).

Table 3-9
Results of Delimiting
with ALL Asterisks

FIELD1 PIC X(8) VALUE IS:	VALUES AFTER UNSTRING OPERATION	
	FIELD2A PIC XXX	FIELD2B PIC XXX JUSTIFIED
ABC*DEF*	ABC	DEF
ABC**DEF	ABC	DEF
A******F	A	F
A*F*****	A	F
A*CDEFG	A	EFG

The next table illustrates the results of an UNSTRING operation that combines ALL with a 2-character delimiter (UNSTRING FIELD1 DELIMITED BY ALL "**" INTO FIELD2A FIELD2B).

Table 3-10
Results of Delimiting with
ALL Double Asterisks

FIELD1 PIC X(8) VALUE IS:	VALUES AFTER UNSTRING OPERATION	
	PIC XXX	PIC XXX JUSTIFIED
ABC**DEF	ABC	DEF
AB**DE**	AB	DE
A***D***	A	*D
A*******	A	*

In addition to unchangeable delimiters, such as literals and figurative constants, delimiters may be designated by identifiers. Identifiers (which may even be subscripted data-names) permit variable delimiting. Consider the following sample statement:

```
UNSTRING FIELD1 DELIMITED BY DEL1
         INTO FIELD2A FIELD2B.
```

Figure 3-27
Delimiting with an Identifier

The data-name, DEL1, must be alphanumeric. It may be a group or elementary item, and it may be edited. (Since the delimiter is not a receiving field, any editing characters will not effect its use, other than contributing to the size of the item.)

If the delimiter contains a subscript, the subscript may be varied as a side effect of the UNSTRING operation. The evaluation of subscripts is discussed later in this section.

3.8.2.1 Multiple Delimiters - The UNSTRING statement has the ability to scan a sending field, searching for a match from a list of delimiters. This list may contain ALL delimiters and delimiters of various sizes. The only requirement of the list is that delimiters must be connected by the word OR.

The following sample statement separates a sending field into three receiving fields. The sending field consists of three strings separated by the following: (1) any number of spaces, or (2) a comma followed by a single space, or (3) a single comma, or (4) a tab character, or (5) a carriage return character. (The ", " must precede the "," in the list if it is ever to be recognized.)

```
UNSTRING FIELD1 DELIMITED BY
         ALL SPACE OR
         ", " OR
         "," OR
         TAB OR
         CR
         INTO FIELD2A FIELD2B FIELD2C.
```

Figure 3-28
Multiple Delimiters

Table 3-11 illustrates the potential of this statement. The tab (represented by the letter t) and carriage return (represented by the letter r) characters represent single character fields containing the ASCII horizontal tab and carriage return characters.

Table 3-11
Results of the Multiple Delimiters
Shown in Figure 3-28

FIELD1 PIC X(12)	FIELD2A PIC XXX	FIELD2B PIC 9999	FIELD2C PIC XXX
A,0,Cr	A	0000	C
At456, E	A	0456	E
A 3 9	A	0003	9
AttBr	A	0000	B
A,,C	A	0000	C
ABCD, 4321,Z	ABC	4321	Z
t--tab character, r--carriage return character			

3.8.3 The COUNT Phrase

The COUNT phrase keeps track of the size of the sending string and stores the length in a user-supplied data area.

The length of a delimited sending field may vary widely (from zero to the full length of the field) and some programs may require knowledge of this length. For example, if it exceeds the size of the receiving field (which is fixed in size) some data may be truncated and the program's logic may require this information.

To use the phrase, simply follow the receiving field name with the words COUNT IN and an identifier. Consider the following sample statement:

```
UNSTRING FIELD1 DELIMITED BY ALL "*"
         INTO FIELD2A COUNT IN COUNT2A
         FIELD2B COUNT IN COUNT2B
         FIELD2C.
```

Figure 3-29
The COUNT Phrase

In this sample statement, the software will count the number of characters between the left-hand end of FIELD1 and the first asterisk in FIELD1 and place that value into COUNT2A; thus, COUNT2A contains the size of the first sending string. The software does not include the delimiter in the count (as it is not a part of the string).

The software then counts the number of characters in the second sending field and places that value into COUNT2B.

The phrase should be used only where needed; in this example the length of the string moved to FIELD2C is not needed, so no COUNT phrase follows it.

If the receiving field is shorter than the value placed in the count field, the software truncates the sending string. (If the number of integer positions in a numeric field is smaller than the value placed into the count field, high-order numeric digits have been lost.) If the software finds a delimiter match on the first character it examines, it places a zero in the count field.

The count field must be described as a numeric integer, either COMP or DISPLAY usage, with no editing symbols nor the character P in its picture-string. The software moves the count value into the count field according to the rules for an elementary numeric MOVE statement

The COUNT phrase may be used only in conjunction with the DELIMITED BY phrase.

3.8.4 The DELIMITER Phrase

The DELIMITER phrase causes the actual character or characters that delimited the sending field to be stored in a user-supplied data area. This phrase is most useful when: (1) the statement contains a delimiter list, (2) any one of the items in the list might have delimited the field, and (3) program logic flow depends on which one found a match. In fact, the DELIMITER and COUNT phrases could be used together and program logic flow could depend on both the size of the sending string and the delimiter character that terminated it.

To use the DELIMITER phrase, simply follow the receiving field name with the words DELIMITER IN and an identifier. (The software places the delimiter character in the area named by the identifier.) Consider the following sample UNSTRING statement:

```
UNSTRING FIELD1 DELIMITED BY "," OR TAB OR
         ALL SPACE OR CR
         INTO FIELD2A DELIMITER IN DELIMA
         FIELD2B DELIMITER IN DELIMB
         FIELD2C.
```

Figure 3-30
The DELIMITER Phrase

After moving the first sending string to FIELD2A, the software takes the character (or characters) that delimited that string and places it in DELIMA. DELIMA, then, contains a comma, or a tab, or a carriage return, or any number of spaces. Since the delimiter string is moved under the rules of the elementary non-numeric MOVE statement, the software truncates or space fills with left or right justification (depending on its data description).

The software then moves the second sending string to FIELD2B and places its delimiting character into DELIMB.

When a sending string is delimited by the end of the sending field (rather than a match on a delimiter) the delimiter string is of zero length. This causes the DELIMITER item to be space filled. The phrase should be used only where needed; in this example, the character that delimits the last sending string is not needed, so no DELIMITER phrase follows FIELD2C.

The data item named in the DELIMITER phrase must be described as an alphanumeric item. It may contain editing characters and it may even be a group item.

When the DELIMITER and COUNT phrases are used together, they must appear in the correct order (DELIMITER phrase preceding the COUNT phrase). Both of the data items named in these phrases may be subscripted or indexed. If they are subscripted, the subscript may be varied as a side effect of the UNSTRING operation. (The evaluation of subscripts is discussed in section 3.8.8.)

3.8.5 The POINTER Phrase

Although the UNSTRING statement normally starts at the left-hand end of the sending field, the POINTER phrase permits the user to select a character position in the sending field for the software to begin scanning. (The scanning, however, remains left-to-right.)

When a sending field is to be dispersed into multiple receiving fields, it often happens that the choice of delimiters, the size of subsequent receiving fields, etc. depend on the value in the first sending string or the character that delimited that string. Thus, the program may need to move the first field, hold its place in the sending field, and examine the results of the operation to determine how to handle the sending items that follow. This is done by using an UNSTRING statement with a POINTER phrase that fills only the first receiving field. When the first string has been moved to a receiving item, the software updates the pointer data item with a new position (one character beyond the delimiter that caused the interruption) to begin the next scanning operation. The program may then examine the new position, the receiving field, the delimiter value, the sending string size, and resume the scanning operation by executing another UNSTRING statement with the same sending field and pointer data item. Thus, the UNSTRING statement can move one sending string at a time, with the form of each move being dependent on the context of the preceding string of data.

The POINTER phrase must follow the last receiving item in the statement. Consider the following two UNSTRING statements with their accompanying POINTER phrases and tests:

```
MOVE 1 TO P.
UNSTRING FIELD1 DELIMITED BY
         ":" OR TAB OR CR OR ALL SPACE
         INTO FIELD2A
         DELIMITER IN DELIMA
         COUNT IN LSIZEA
         WITH POINTER PNTR.
IF LSIZEA = 0 GO TO NO-LABEL-PROCESS.
IF DELIMA =  ":"
         IF PNTR > 8 GO TO BIG-LABEL-PROCESS
         ELSE GO TO LABEL-PROCESS.
IF DELIMA = TAB GO TO BAD-LABEL PROCESS.

     ...

UNSTRING FIELD1 DELIMITED BY ...  WITH POINTER PNTR.
```

Figure 3-31
The POINTER Phrase

PNTR contains the current position of the scanner in the sending field. The second UNSTRING statement uses PNTR to begin scanning the additional sending strings in FIELD1.

Since the software considers the left-most character to be character position one, the value returned by PNTR may be used to examine the next character. To do this, simply use PNTR as a subscript on the sending field (providing that the sending field is also described as a table of characters). For example, consider the following sample coding:

```
01 FIELD1.
   02 FIELD1-CHAR OCCURS 40 TIMES.

     ...

   UNSTRING FIELD1
        ...
        WITH POINTER PNTR.
   IF FIELD1-CHAR(PNTR) = "X" ...
```

Figure 3-32
Examining the Next Character
By Using the Pointer Data
Item as a Subscript

Another way to examine the next character of the sending field is to use the UNSTRING statement to move it to a 1-character receiving field. Consider the sample coding in figure 3-33.

```
UNSTRING FIELD1
         ...
         WITH POINTER PNTR.
UNSTRING FIELD1 INTO CHAR1 WITH POINTER PNTR.
SUBTRACT 1 FROM PNTR.
IF CHAR1 = "X" ...
```

Figure 3-33
Examining the Next Character
By Placing It Into a 1-Character Field

The program must decrement PNTR in order for this case to work like the one illustrated in Figure 3-32, since the second UNSTRING statement will increment the pointer value by 1.

The program must initialize the POINTER phrase data item before the UNSTRING statement uses it. The software will terminate the UNSTRING operation if the initial value of the pointer is less than one or greater than the length of the sending field. (A pointer value that is less than one or greater than the length of the sending field causes an overflow condition. Overflow conditions are discussed in section 3.8.7.)

The POINTER and TALLYING phrases may be used together in the same UNSTRING statement: but, when both are used, the POINTER phrase must precede the TALLYING phrase.

3.8.6 The TALLYING Phrase

The TALLYING phrase counts the number of receiving fields that received data from the sending field.

When an UNSTRING statement contains several receiving fields, the possibility exists that there may not always be as many sending strings as there are receiving fields. The TALLYING phrase provides a convenient method for keeping a count of how many fields were acted upon.

```
MOVE 0 TO RCOUNT.
UNSTRING FIELD1 DELIMITED BY "," OR ALL SPACE
         INTO FIELD2A
              FIELD2B
              FIELD2C
              FIELD2D
              FIELD2E
              TALLYING IN RCOUNT.
```

Figure 3-34
The TALLYING Phrase

If the software has moved only three sending strings when it reaches the end of FIELD1, it adds 3 to RCOUNT. The first three fields (FIELD2A, FIELD2B, and FIELD2C) contain data from the operation, and the last two (FIELD2D and FIELD2E) do not.

The TALLYING data item always contains the sum of its initial contents plus the number of sending strings acted upon by the UNSTRING command just executed. Thus, the programmer may want to initialize the tally count before each use.

When used in the same statement with a POINTER phrase, the TALLYING phrase must follow the POINTER phrase and both phrases must follow all of the field names, the DELIMITER and COUNT phrases. The data items for both phrases must contain numeric integers, that is, be without editing characters or the letter P in their picture-strings; both data items may be either COMP or DISPLAY usage. They may be signed or unsigned and, if they are DISPLAY usage, they may contain any desired sign option.

The data items for both phrases may be subscripted or indexed, or they may be used as subscripts on other fields in the statement. (The evaluation of subscripts is discussed in section 3.8.8.) A convenient use of the TALLYING phrase data item is as a subscript of a receiving field. Consider the following sample coding, which causes program control to execute the UNSTRING statement repeatedly until it exhausts the sending field.

```
     MOVE 1 TO PNTR, TLY.
PAR1.  UNSTRING FIELD1 DELIMITED BY "," OR CR
                INTO FIELD2(TLY)
                DELIMITER IN DEL2
                WITH POINTER PNTR
                TALLYING IN TLY.
     IF DEL2 = "," GO TO PAR1.
```

Figure 3-35
The POINTER and TALLYING Phrases
Used Together

This sample coding causes program control to loop through the UNSTRING statement, using the pointer, PNTR, to scan across FIELD1 with successive executions. Each comma isolates a sending string until control reaches either a carriage return character or the end of FIELD1. If it reaches the end of the field without encountering a carriage return character, the software places a space into the delimiter field, DEL2, and control falls through the IF statement and out of the loop.

Since the TALLYING data item, TLY, is increased by 1 after each data movement, it serves as a subscript on the receiving field. In effect this causes the software to unpack the value in FIELD1 into an array of fixed-size fields. Further, an array of COUNT data items can be supplied and loaded by the UNSTRING/TALLYING statement by adding the COUNT IN phrase to the coding in Figure 3-35, as is shown in Figure 3-36.

```
COUNT IN C(TLY)
```

Figure 3-36
Subscripting the COUNT Phrase
With the TALLYING Data Item

The TALLYING data item, in the above example, is one greater than the number of receiving fields acted upon by the UNSTRING operation. This is because the data item must be initialized to a value of one in order to be used as a subscript for the first receiving item.

3.8.7 The OVERFLOW Phrase

The OVERFLOW phrase detects the overflow condition and provides an imperative statement to be executed when it detects the condition. An overflow condition exists when either of the following two situations occurs:

1. The UNSTRING statement is about to be executed and its pointer data item contains a value of less than one or greater than the size of the sending field. When it detects this situation, the software executes the OVERFLOW phrase before it moves any data. Thus, the values of all of the receiving fields remain unchanged.

2. The UNSTRING statement has filled all of the receiving fields and data still remains in the sending field that has not been matched as a delimiter or included in a sending string. When it detects this situation, the software executes the OVERFLOW phrase after it has executed the UNSTRING statement. Thus, the values of all of the receiving fields are updated, but some data has not been moved.

If the UNSTRING operation causes the scanner to move off the end of the sending field (thus exhausting it), the software will not execute the OVERFLOW phrase.

Consider the following set of instructions, which cause program control to execute the UNSTRING statement repeatedly until it exhausts the sending field. The TALLYING data item is a subscript indexing the receiving field. (Compare this loop with the one in Figure 3-35, which accomplishes the same thing.)

```
        MOVE 1 TO TLY PNTR.
PAR1.   UNSTRING FIELD1 DELIMITED BY "," OR CR
                 INTO FIELD2(TLY)
                 WITH POINTER PNTR
                 TALLYING IN TLY
                 ON OVERFLOW GO TO PAR1.
```

Figure 3-37
Using the OVERFLOW Phrase

NOTE

The overflow condition also occurs if the value of a pointer data item lies outside the sending field at the start of execution of the UNSTRING statement. (The pointer value must not be less than 1, nor greater than the length of the sending field.) This type of overflow is not distinguishable from the overflow condition described at the start of this section, except that this condition causes the UNSTRING statement to terminate before any data movement takes place. Then, the values of all receiving fields remain unchanged.

3.8.8 Subscripted Fields in UNSTRING Statements

Since the flexibility of the UNSTRING statement is enhanced by subscripting and indexing and particularly by subscripting with other fields within the statement (such as subscripting the receiving field with the TALLYING data item as discussed above), it is important to understand how often and exactly when the software evaluates these subscripts and indexes. This sub-section discusses the frequency and times of subscript evaluation.

The software evaluates subscripts and indexes on the following items only once, at the initiation of the UNSTRING statement; thus, any change in subscript values during the execution of the statement has no effect on these fields:

1. Sending field,
2. POINTER data item,
3. TALLYING data item.

The software evaluates subscripts and indexes on the following items immediately before it moves data into the item. It moves the data to these items in the order in which they are listed in the statement (which is the same order as below):

1. Receiving field,
2. DELIMITER data item,
3. COUNT data item.

The software evaluates any subscripts and indexes on the data-names in the DELIMITED BY phrase (delimiters) immediately before it scans each sending string looking for a delimiter match. Thus, it re-evaluates these data-names once for each receiving field in the statement.

If any of the following items are used as subscripts on any receiving fields, the programmer must be aware of the point at which these items are updated:

- POINTER data-item,
- TALLYING data-item,
- COUNT data-item,
- Another receiving field.

Figure 3-38 illustrates, with a flow chart, the sequence of evaluation operations:

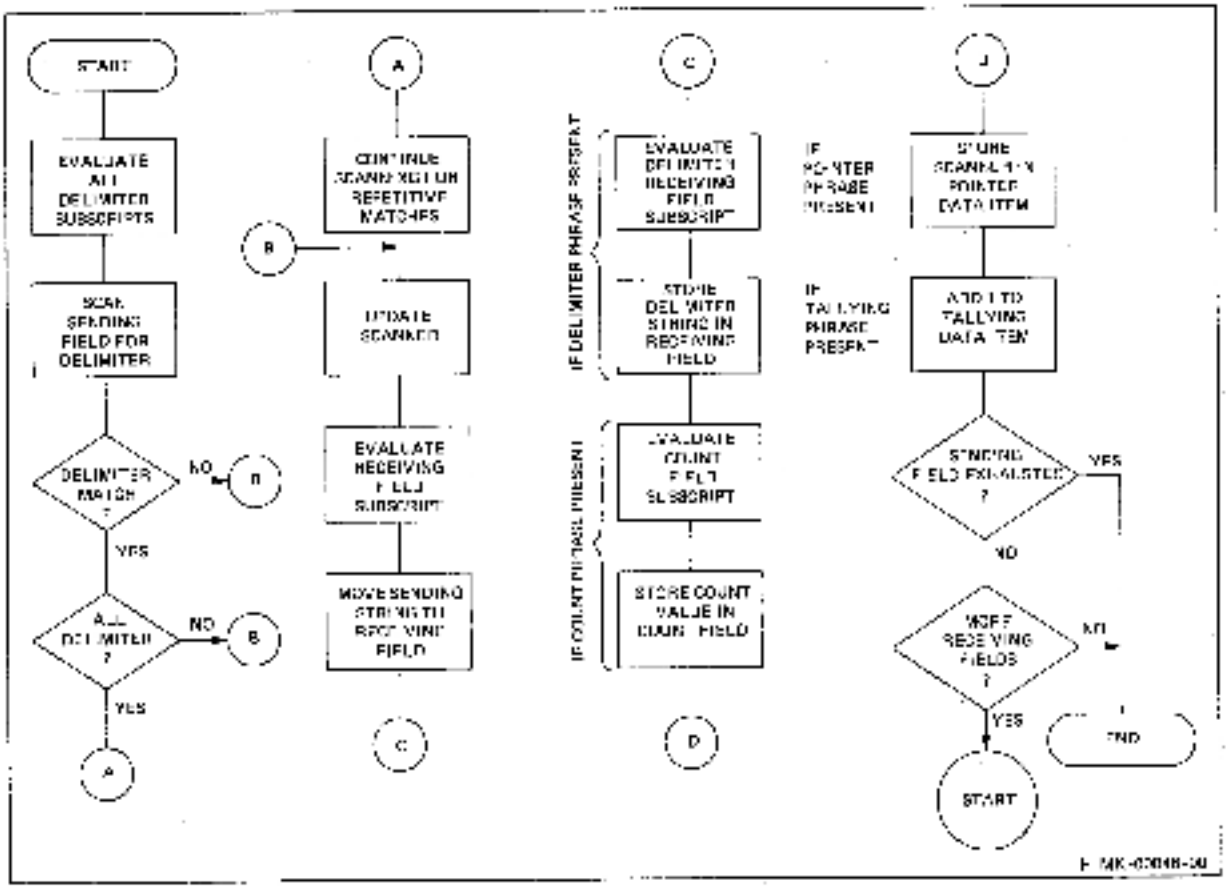


Figure 3-38
Sequence of Subscript Evaluation

NOTE

The rules in this section concerning the exact point at which the software evaluates the identifiers in the DELIMITED BY phrase and the point at which it updates the POINTER and TALLYING data items, are rules that are specified by 1974 American National Standard COBOL, as opposed to the STRING statement where these are not so specified.

3.8.9 Common Errors, UNSTRING Statement

The most common errors made when writing UNSTRING statements are:

- Leaving the OR connector out of a delimiter list;
- Misspelling or interchanging the words, DELIMITED and DELIMITER;
- Writing the DELIMITER and COUNT phrases in the wrong order when both are present (DELIMITER must precede COUNT);
- Leaving out the word INTO or writing it as TO;
- Repeating the word INTO where it is not needed; thus:

```
UNSTRING FIELD1 DELIMITED BY SPACE OR TAB
         INTO FIELD2A DELIMITER IN DELIMA
         INTO FIELD2B DELIMITER IN DELIMB
         INTO FIELD2C DELIMITER IN DELIMC.
```

Figure 3-39
Erroneously Repeating the Word INTO

- Writing the POINTER and TALLYING phrases in the wrong order (POINTER must precede TALLYING).

3.9 THE INSPECT STATEMENT

The INSPECT statement examines the character positions in a field and counts or replaces certain characters (or groups of characters) in that field.

Like the STRING and UNSTRING operations, INSPECT operations scan across the field from left to right; further, like those two statements, the INSPECT statement features a phrase which allows it to begin or terminate the scanning operation with a delimiter match. (Thus, the operation can begin within the field instead of at the left-hand end, or it may begin at the left-hand end and terminate within the field.)

The TALLYING operation (which counts certain characters in the field) and the REPLACING operation (which replaces certain characters in the field) are quite versatile and may be applied to all of the characters in the delimited area of the field being inspected, or they may be applied only to those characters that match a given character string under stated conditions. Consider the following sample statements, which both cause a scan of the complete field:

```
INSPECT FIELD1 TALLYING TLY FOR ALL "B".
```

Figure 3-40
Sample INSPECT...TALLYING Statement

This statement scans FIELD1 looking for the character B. Each time it finds a B, it increments TLY by 1.

```
INSPECT FIELD1 REPLACING ALL SPACE BY ZERO.
```

Figure 3-41
Sample INSPECT...REPLACING Statement

This statement scans FIELD1 looking for space characters. Wherever it finds a space character, it replaces it with zero.

One INSPECT statement can contain both a TALLYING phrase and a REPLACING phrase. However, when used together, the TALLYING phrase must precede the REPLACING phrase. An INSPECT statement with both phrases is equivalent to two separate INSPECT statements and, in fact, the software compiles such a statement into two distinct INSPECT statements. (To simplify debugging, therefore, it is best to initially write the two phrases in separate INSPECT statements.)

3.9.1 The BEFORE/AFTER Phrase

The BEFORE/AFTER phrase acts as a delimiter and (possibly) restricts the area of the field being inspected.

The following sample statement would count only the zeroes that precede the percent sign (%) in FIELD1.

```
INSPECT FIELD1 TALLYING TLY
        FOR ALL ZEROES BEFORE "%".
```

Figure 3-42
Sample INSPECT...BEFORE Statement

The delimiter (the percent sign in the preceding sample statement) can be a single character, a string of characters, or any figurative constant. Further, it can be either an identifier or a literal.

- If the delimiter is an identifier, it must be an elementary data item of DISPLAY usage. It may be alphabetic, alphanumeric, or numeric, and, it may contain editing characters. The compiler always treats the item as if it had been described as an alphanumeric string. (It does this by implicit redefinition of the item, as described in Section 3.9.2.)

- If the delimiter is a literal, it must be non-numeric.

The software repeatedly compares the delimiter characters against an equal number of characters in the field being inspected. If none of the characters matches the delimiter, or if insufficient characters remain in the field for a full comparison (at the right-hand end), the software considers the comparison to be unequal.

The examples of the INSPECT statement in Figure 3-43, illustrate the way the delimiter character finds a match in the field being inspected. (The portion of the field the statement ignores as a result of the BEFORE/AFTER phrase delimiters is crossed out with a slash, and the portion it inspects is underlined.)

INSTRUCTION	FIELD1 VALUE
INSPECT FIELD1...BEFORE "E". INSPECT FIELD1...AFTER "E".	ABCDEFGHI ABCDEFGHI
INSPECT FIELD1...BEFORE "R". INSPECT FIELD1...AFTER "R".	ABCDEFGHI ABCDEFGHI
INSPECT FIELD1...BEFORE "AB". INSPECT FIELD1...AFTER "AB".	ABCDEFGHI ABCDEFGHI
INSPECT FIELD1...BEFORE "HI". INSPECT FIELD1...AFTER "HI".	ABCDEFGHI ABCDEFGHI
INSPECT FIELD1...BEFORE "I ". INSPECT FIELD1...AFTER "I ".	ABCDEFGHI ABCDEFGHI
The ellipsis represents the position of the TALLYING or REPLACING phrase.	

Figure 3-43
Matching the Delimiter Characters
to the Characters in a Field

The software scans the field for a delimiter match before it scans for the inspection operation (TALLYING or REPLACING), thus establishing the limits of the operation before beginning the actual inspection. Section 3.9.3 further discusses the importance of the separate scan.

3.9.2 Implicit Redefinition

The software requires that certain fields referred to by the INSPECT statement be alphanumeric fields. If one of these fields was described as another data class, the compiler redefines that field so the INSPECT statement can handle it as a simple alphanumeric string. This implicit redefinition is conducted as follows:

- If the field was described as alphabetic, alphanumeric edited, or unsigned numeric, the compiler simply redefines it as alphanumeric. This is a compile-time operation; no data movement occurs at run time.

- If the field was described as signed numeric, the compiler first removes the sign and then redefines the field as alphanumeric. If the sign is a separate character, the compiler ignores that character, essentially shortening the field, and that character does not participate in the implicit redefinition. If the sign is an "overpunch" on the leading or trailing digit, the compiler actually removes the sign value and leaves the character with only the numeric value that was stored in it. The compiler alters the digit position containing the sign before beginning the INSPECT operation and restores it to its former value after the operation. If the sign's digit position does not contain a valid ASCII signed numeric digit, the action of the redefinition causes the value to change. Table 3-12 shows these original, altered, and restored values.

The compiler never moves an implicitly redefined item from its storage position. All redefinition occurs in place.

The position of an implied decimal point on numeric quantities does not affect implicit redefinition.

Table 3-12
Original, Altered, and Restored Values Resulting from Implicit Redefinition

ORIGINAL VALUE	ALTERED VALUE	RESTORED VALUE
{ (7B)	0 (30)	{ (7B)
A (41)	1 (31)	A (41)
B (42)	2 (32)	B (42)
C (43)	3 (33)	C (43)
D (44)	4 (34)	D (44)
E (45)	5 (35)	E (45)
F (46)	6 (36)	F (46)
G (47)	7 (37)	G (47)
H (48)	8 (38)	H (48)
I (49)	9 (39)	I (49)
} (7D)	0 (30)	} (7D)
J (4A)	1 (31)	J (4A)
K (4B)	2 (32)	K (4B)
L (4C)	3 (33)	L (4C)
M (4D)	4 (34)	M (4D)
N (4E)	5 (35)	N (4E)
O (4F)	6 (36)	O (4F)
P (50)	7 (37)	P (50)
Q (51)	8 (38)	Q (51)
R (52)	9 (39)	R (52)
0 (30)	0 (30)	{ (7B)
1 (31)	1 (31)	A (41)
2 (32)	2 (32)	B (42)
3 (33)	3 (33)	C (43)
4 (34)	4 (34)	D (44)
5 (35)	5 (35)	E (45)
6 (36)	6 (36)	F (46)
7 (37)	7 (37)	G (47)
8 (38)	8 (38)	H (48)
9 (39)	9 (39)	I (49)
All other values	0 (30)	{ (7B)

3.9.3 The INSPECT Operation

Regardless of the type of inspection (TALLYING or REPLACING), the INSPECT statement has only one method for inspecting the characters in the field. This section describes this method.

However, before discussing how the inspection operation is conducted, let's analyze the INSPECT statement itself:

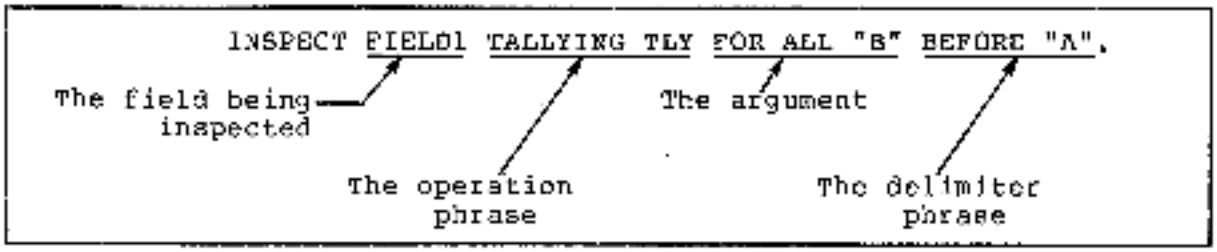


Figure 3-44
Sample INSPECT Statement

The format of the INSPECT statement requires that a field be named which is to be inspected (FIELD1 above); the field name must be followed by an operation phrase (TALLYING TLY above); and, that phrase must be followed by one or more identifiers or literals ("B" above). These identifiers or literals comprise the "arguments" (items to be compared to the field being inspected). More than one argument makes up the "argument list".

- TALLYING Arguments

  Each argument in an argument list can have other fields associated with it. Thus, each argument that is used in a TALLYING operation must have a tally counter (TLY above) associated with it. The software increments the tally counter each time it matches the argument with a character or group of characters in the field being inspected.

- REPLACING Arguments

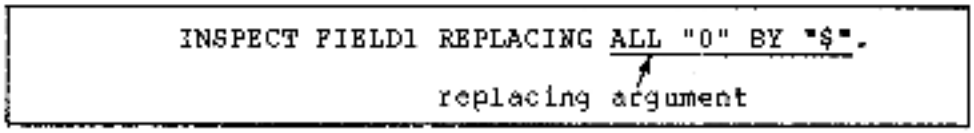


Figure 3-45
Sample REPLACING Argument

  Each argument in an argument list that is used in a REPLACING operation must have a replacement item ($ above) associated with it. The software uses the replacement item to replace each string of characters in the field that matches the argument.

Each argument in an argument list (that is used with either a TALLYING or REPLACING operation) may have a delimiter field (BEFORE/AFTER phrase) associated with it. If the delimiter field is not present, the software applies the argument to the entire field. If the delimiter field is present, the software applies the argument only to that portion of the field specified by the BEFORE/AFTER phrase.

3.9.3.1 Setting the Scanner - The INSPECT operation begins by setting the scanner to the leftmost character position of the field being inspected. It remains on this character until an argument has been matched with a character (or characters) or until all arguments have failed to find a match at that position.

3.9.3.2 Active/Inactive Arguments - When an argument has a BEFORE/AFTER phrase associated with it, that argument has a delimiter and may not be eligible to participate in a comparison at every position of the scanner. Thus, each argument in the argument list has an active/inactive status at any given setting of the scanner.

For example, an argument that has an AFTER phrase associated with it starts the INSPECT operation in an inactive state. The delimiter of the AFTER phrase must find a match before the argument can participate in the comparison. When the delimiter finds a match, the software retains the character position beyond the matched character string; then, when the scanner reaches or passes this position, the argument becomes active.

```
INSPECT FIELD1 TALLYING TLY
        FOR ALL "B" AFTER "X".
```

Figure 3-46
Sample AFTER Delimiter Phrase

If FIELD1 in Figure 3-46 has a value of "ABABXZBA", the argument B remains inactive until the scanner finds a match for the delimiter X. Thus, argument B remains inactive while the software scans character positions 1 through 5. At character position 5, the delimiter X finds a match, and since the character position beyond the matched delimiter character is the point at which the argument becomes active, argument B is compared for the first time at character position 6. It finds a successful match at character position 7 and this causes TLY to be incremented by 1.

The examples in Figure 3-47 illustrate other situations where the arguments and/or the delimiters are longer than one character. (Consider the sample statement to be an INSPECT...TALLYING statement that is scanning FIELD1, tallying in TLY, and looking for the arguments and delimiters in the left-hand column. Assume that TLY is initialized to 0.)

ARGUMENT AND DELIMITER	FIELD1 VALUE	ARGUMENT ACTIVE AT POSITION	CONTENTS OF TLY AFTER SCAN
"B" AFTER "XX"	BXBXXXXBB	6	2
	XXXXXXXX	3	0
	BXBXBBBBXX	never	0
"X" AFTER "XX"	BXBXXBXXB	6	2
	XXXXXXXX	3	6
	BBBBBBXX	never	0
"B" AFTER "XB"	BXYBXBXX	7	0
	XBXBXBXB	3	3
	BBBBBBXB	never	0
"BX" AFTER "XB"	XXXXBXXXX	6	0
	XXXXBBXXX	6	1
	XXBXXXXBX	4	1

Figure 3-47
Where Arguments Become Active in a Field

When an argument has an associated BEFORE delimiter, the inactive/active states reverse roles: the argument is in an active state when the scanning begins, and becomes inactive at the character position that matches the delimiter. Additionally, regardless of the presence of the BEFORE delimiter, an argument becomes inactive when the scanner approaches the right-hand end of the field and the remaining characters are fewer in number than the characters in the argument. (In such a case, the argument cannot possibly find a match in the field so it becomes inactive.)

Since the BEFORE/AFTER delimiters are found on a separate scan of the field, the software recognizes and sets up the delimiter boundaries before it scans for an argument match; therefore, the same characters can be used as arguments and delimiters in the same phrase.

3.9.3.3 **Finding an Argument Match** - The software selects arguments from the argument list in the order in which they appear in the list. If the first one it selects is an active argument and the conditions stated in the INSPECT statement allow a comparison, the software compares it to the character at the position of the scanner. If the active argument does not find a match, the software takes the next active argument from the list and compares that to the same character. If none of the active arguments finds a match, the scanner moves one position to the right and begins the inspection operation again with the first active argument in the list. The inspection operation terminates at the right-hand end of the field.

When an active argument does find a match, the software ignores any remaining arguments in the list and conducts the TALLYING or REPLACING operation on the character. The scanner moves to a new position and the next inspection operation begins with the first argument in the list. (The INSPECT statement may contain additional conditions, which are described later in this section; this discussion, however, assumes that the argument match is allowed to take place and that inspection is allowed to continue following the match.)

The software updates the scanner by adding the size of the matching argument to it. This moves the scanner to the next character beyond the string of characters that matched the argument. Thus, once an active argument matches a string of characters, the statement does not inspect those character positions again unless program control executes the entire statement again.

3.9.4 Subscripted Fields in INSPECT Statements

Any identifier named in an INSPECT statement may be subscripted or indexed.

The software evaluates all subscripts in an INSPECT statement once, before the inspection begins; therefore, if the action of the INSPECT statement alters one of the subscripts, the new subscript value has no effect on the selection of operands during that inspection operation. For example, consider the following illustration:

```
MOVE 1 TO TLY.
INSPECT FIELD1 TALLYING TLY
        FOR ALL X(TLY).
```

Figure 3-48
Sample Subscripted Argument

In this sample statement, the software evaluates the address of X(TLY) only once, before it begins inspecting the field; hence, it will evaluate X(TLY) as X(1). The alteration of TLY by the action of inspecting and tallying has no effect on the choice of the X operand. (X(1) will be used throughout the operation.)

NOTE

When subscripting an INSPECT statement that contains both a TALLYING and a REPLACING phrase, keep in mind that the statement will be compiled into two separate INSPECT statements. Therefore, any field that is altered by the action of the INSPECT...TALLYING statement will be in its altered state if used as a subscript by the INSPECT...REPLACING statement.

3.9.5 The TALLYING Phrase

An INSPECT statement that contains a TALLYING phrase counts the occurrence of various character strings under certain stated conditions. It keeps the count in a user-designated field called, here, a tally counter.

3.9.5.1 The Tally Counter - The identifier that follows the word TALLYING designates the tally counter. The identifier may be subscripted or indexed. The data item must be a numeric integer with no editing or P characters; it may be COMP or DISPLAY usage, and it may be signed (separate or overpunched).

Each time the tally argument matches the delimited string being inspected, the software adds 1 to the tally counter.

The programmer can initialize the tally counter to any numeric value. (The INSPECT statement does not initialize it.)

3.9.5.2 The Tally Argument - The tally argument specifies a character-string and a condition under which that string should be compared to the delimited string being inspected. The following figure shows the format of the tally argument:

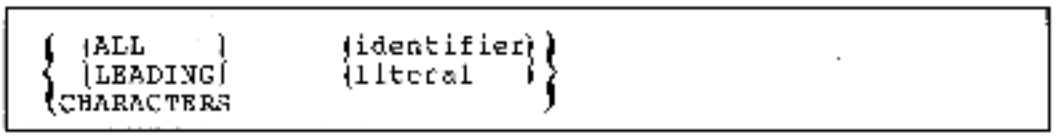


Figure 3-49
Format of the Tally Argument

The CHARACTERS form of the tally argument specifies that every character in the delimited string being inspected should be considered to match an imaginary character that serves as the tally argument. This increments the tally counter by a value that equals the size of the delimited string. For example, the statement in the following illustration causes TLY to be incremented by the number of characters that precede the first comma, regardless of what those characters might be.

```
INSPECT FIELD1 TALLYING TLY FOR
        CHARACTERS BEFORE ",".
```

Figure 3-50
CHARACTERS Form of the Tally Argument

When an active argument does find a match, the software ignores any remaining arguments in the list and conducts the TALLYING or REPLACING operation on the character. The scanner moves to a new position and the next inspection operation begins with the first argument in the list. (The INSPECT statement may contain additional conditions, which are described later in this section; this discussion, however, assumes that the argument match is allowed to take place and that inspection is allowed to continue following the match.)

The software updates the scanner by adding the size of the matching argument to it. This moves the scanner to the next character beyond the string of characters that matched the argument. Thus, once an active argument matches a string of characters, the statement does not inspect those character positions again unless program control executes the entire statement again.

3.9.4 Subscripted Fields in INSPECT Statements

Any identifier named in an INSPECT statement may be subscripted or indexed.

The software evaluates all subscripts in an INSPECT statement once, before the inspection begins; therefore, if the action of the INSPECT statement alters one of the subscripts, the new subscript value has no effect on the selection of operands during that inspection operation. For example, consider the following illustration:

```
MOVE 1 TO TLY.
INSPECT FIELD1 TALLYING TLY
        FOR ALL X(TLY).
```

Figure 3-48
Sample Subscripted Argument

In this sample statement, the software evaluates the address of X(TLY) only once, before it begins inspecting the field; hence, it will evaluate X(TLY) as X(1). The alteration of TLY by the action of inspecting and tallying has no effect on the choice of the X operand. (X(1) will be used throughout the operation.)

NOTE

When subscripting an INSPECT statement that contains both a TALLYING and a REPLACING phrase, keep in mind that the statement will be compiled into two separate INSPECT statements. Therefore, any field that is altered by the action of the INSPECT...TALLYING statement will be in its altered state if used as a subscript by the INSPECT...REPLACING statement.

3.9.5 The TALLYING Phrase

An INSPECT statement that contains a TALLYING phrase counts the occurrence of various character strings under certain stated conditions. It keeps the count in a user-designated field called, here, a tally counter.

3.9.5.1 The Tally Counter - The identifier that follows the word TALLYING designates the tally counter. The identifier may be subscripted or indexed. The data item must be a numeric integer with no editing or P characters; it may be COMP or DISPLAY usage, and it may be signed (separate or overpunched).

Each time the tally argument matches the delimited string being inspected, the software adds 1 to the tally counter.

The programmer can initialize the tally counter to any numeric value. (The INSPECT statement does not initialize it.)

3.9.5.2 The Tally Argument - The tally argument specifies a character-string and a condition under which that string should be compared to the delimited string being inspected. The following figure shows the format of the tally argument:

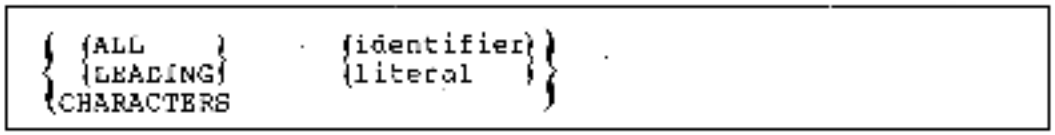


Figure 3-49
Format of the Tally Argument

The CHARACTERS form of the tally argument specifies that every character in the delimited string being inspected should be considered to match an imaginary character that serves as the tally argument. This increments the tally counter by a value that equals the size of the delimited string. For example, the statement in the following illustration causes TLY to be incremented by the number of characters that precede the first comma, regardless of what those characters might be.

```
INSPECT FIELD1 TALLYING TLY FOR
        CHARACTERS BEFORE ",".
```

Figure 3-50
CHARACTERS Form of the Tally Argument

Thus, the first argument is initially inactive and becomes active only after the scanner encounters an A; the second argument begins the scan in the active state but becomes inactive after a B has been encountered; and the third argument is active during the entire scan of FIELD1.

Figure 3-55 shows various values of FIELD1 and the contents of the three tally counters after the scan. Assume that the counters are initialized to 0 before the INSPECT statement.

FIELD1 VALUE	CONTENTS OF TALLY COUNTERS AFTER SCAN		
	T1	T2	T3
A.C;D.E,F	1	2	1
A.B.C.D	0	1	0
A,B,C,D	3	0	0
A;B;C;D	0	0	3
*,B,C,D	0	0	0

Figure 3-55
Results of the Scan in Figure 3-54

The BEFORE/AFTER phrase applies only to the argument that precedes it, and delimits the field for that argument only. Each BEFORE/AFTER phrase causes a separate scan of the field to determine the limits of the field for its corresponding argument.

3.9.5.4 **Interference in Tally Argument Lists** - When several tally arguments contain one or more identical characters that are active at the same time, they may interfere with each other (i.e., when one of the arguments finds a match, the scanner is stepped past the matching character(s) which prevents those character(s) from being considered for any other match).

The example in Figure 3-56 illustrates two identical tally arguments that do not interfere with each other since they are not active at the same time. (The first A in FIELD1 causes the first argument to become inactive and the second argument to become active.)

```
MOVE 0 TO T1 T2.
INSPECT FIELD1 TALLYING
        T1 FOR ALL "," BEFORE "A"
        T2 FOR ALL "," AFTER "A".
```

Figure 3-56
Two Tallying Arguments that
Do Not Interfere with Each Other

The two identical tally arguments in Figure 3-57 will interfere with each other since both are active at the same time. (For any given position of the scanner, the arguments are applied to FIELD1 in the order in which they appear in the statement. When one of them finds a match, the scanner moves to the next position and ignores the

remaining arguments in the argument list.) Each comma in FIELD1 causes T1 to be incremented by 1 and the second argument to be ignored. Thus, T1 will always contain an accurate count of all of the commas in FIELD1, and T2 will always be unchanged.

```
INSPECT FIELD1 TALLYING
        T1 FOR ALL ","
        T2 FOR ALL "," AFTER "A".
```

Figure 3-57
Two Tallying Arguments that
Do Interfere with Each Other

The following statement achieves the same results as the statement in Figure 3-56. The first argument does not become active until the scanner encounters an A. The second argument tallies all commas that precede the A. After the A, the first argument counts all commas and causes the second argument to be ignored. Thus, T1 contains the number of commas that precede the first A and T2 contains the number of commas that follow the first A. This statement works well as written, but could be more confusing to debug than the one in Figure 3-56.

```
INSPECT FIELD1 TALLYING
        T2 FOR ALL "," AFTER "A"
        T1 FOR ALL ",".
```

Figure 3-58
Two Tallying Arguments that,
Because of their Positioning,
Only Partially Interfere with
Each Other

The preceding three examples show that one INSPECT statement cannot count any character more than once. Thus, when using the same character in more than one argument of an argument list, consider the nature of the interference and choose the order of the arguments very carefully. The solution to the problem may require two or more INSPECT statements. Consider the following problem:

```
INSPECT FIELD1 TALLYING
        T1 FOR ALL "AB"
        T2 FOR ALL "BC".
```

Figure 3-59
An Attempt to Tally the Character B
with Two Arguments

If FIELD1 contains "ABCABC", after the scan T1 will be incremented by a 2 and T2 will be unaltered. The successful matching of the argument includes each B in the field. Each match resets the scanner to the character position to the right of the B, and causes the second

argument to never be successfully matched. Reversing the order of the arguments has no effect, the results remain the same. Only separate INSPECT statements can develop the desired counts.

Sometimes the programmer can use the interference characteristics of the INSPECT statement to good advantage. Consider the following sample argument list:

```
MOVE 0 TO T4 T3 T2 T1.
INSPECT FIELD1 TALLYING
        T4 FOR ALL "****"
        T3 FOR ALL "***"
        T2 FOR ALL "**"
        T1 FOR ALL "*".
```

Figure 3-60
Tallying Asterisk Groupings

The argument list in Figure 3-60 counts all of the asterisks in FIELD1 but in four different tally counters. T4 counts the number of times that four asterisks occur together; T3 counts the number of times three asterisks appear together; T2 counts double asterisks; and T1 counts singles.

If FIELD1 contains a string of more than four consecutive asterisks, the argument list breaks the string into groups of four, and counts them in T4. It then counts the less-than-four remainder in T3, T2, or T1.

Reversing the order of the arguments in this list causes T1 to count all of the asterisks and T2, T3, and T4 to remain unchanged.

When the LEADING condition is used with an argument in the argument list, that argument becomes inactive as soon as it fails to be matched in the field being inspected. Therefore, when two arguments in an argument list contain one or more identical characters and one of the arguments has a LEADING condition, the argument with the LEADING condition should appear first. Consider the following sample statement:

```
MOVE 0 TO T1 T2.
INSPECT FIELD1 TALLYING
        T1 FOR LEADING "*"
        T2 FOR ALL "*".
```

Figure 3-61
Placing the LEADING Condition
in the Argument List

The placement of the LEADING condition in this sample statement causes T1 to count only leading asterisks in FIELD1; the occurrence of any other character stops this counting and causes the first tally argument to become inactive. T2 keeps a count of any remaining asterisks in FIELD1.

Reversing the order of the arguments in this statement results in an argument list that can never increment T1.

```
INSPECT FIELD1 TALLYING
        T2 FOR ALL "*"
        T1 FOR LEADING "*".
```

Figure 3-62
Reversing the Argument
List in Figure 3-61

If the first character in FIELD1 is not an asterisk, neither argument can match it and the second argument becomes inactive. If the first character in FIELD1 is an asterisk, the first argument matches and causes the second argument to be ignored. The first non-asterisk character in FIELD1 will fail to match the first argument and the second argument will become inactive. (The second argument becomes inactive because it has not found a match in all of the preceding characters.)

An argument with both a LEADING condition and a BEFORE phrase can sometimes sucessfully "delimit" the field being inspected:

```
MOVE 0 TO T1 T2.
INSPECT FIELD1 TALLYING
     T1 FOR LEADING SPACES
     T2 FOR ALL "   " BEFORE "."
     T2 FOR ALL "  " BEFORE "."
     T2 FOR ALL " " BEFORE ".".
IF T2 > 0 ADD 1 TO T2.
```

Figure 3-63
An Argument List that Counts
Words in a Statement

The statements in Figure 3-63 count the number of "words" in the English statement in FIELD1. (This assumes that no more than three spaces separate the words in the sentence and that the sentence ends with a period.) When FIELD1 has been scanned, T2 contains the number of gaps between the words. Since a count of the gaps renders a number that is one less than the number of words, the conditional statement adds one to the count.

The first argument removes any leading spaces, counting them in a different tally counter. This shortens FIELD1 by preventing the application of the second through the fourth arguments until the scanner finds a non-space character. The BEFORE phrase on each of the other arguments causes them to become inactive when the scanner reaches the period at the end of the sentence. Thus, the BEFORE phrases "shorten" FIELD1 by making the second through the fourth arguments inactive before the scanner reaches the right-hand end of FIELD1. If the sentence in FIELD1 is indented with tab characters instead of spaces, a second LEADING argument can count the tab characters. Figure 3-64 illustrates this technique:

```
INSPECT FIELD1 TALLYING
        T1 FOR LEADING SPACES
        T1 FOR LEADING TAB
        T2 FOR ALL "    " etc.
```

Figure 3-64
Counting Leading Tab or Space Characters

When an argument list contains a CHARACTERS argument, it should be the last argument in the list. Since the CHARACTERS argument always matches the field, it prevents the application of any of the following arguments in the list. However, as the last argument in an argument list, it can count the remaining characters in the field being inspected. Consider the following illustration.

```
MOVE 0 TO T1 T2 T3 T4 T5.
INSPECT FIELD1 TALLYING
        T1 FOR LEADING SPACES
        T2 FOR ALL "." BEFORE ","
        T3 FOR ALL "+" BEFORE ","
        T4 FOR ALL "-" BEFORE ","
        T5 FOR CHARACTERS BEFORE ",".
```

Figure 3-65
Counting the Remaining Characters
With the CHARACTERS Argument

If FIELD1 is known to contain a number in the form frequently used to input data, it may contain a plus or minus sign, and a decimal point; further, the number may possibly be preceded by spaces and terminated by a comma. If this statement were compiled and executed, it would deliver the following results:

T1 would contain the number of leading spaces,

T2 would contain the number of periods,

T3 would contain the number of plus signs,

T4 would contain the number of minus signs,

T5 would contain the number of remaining characters (assumed to be numeric), and

the sum of T1 through T5 (plus 1) gives the character position occupied by the terminating comma.

3.9.6 The REPLACING Phrase

When an INSPECT statement contains a REPLACING phrase, that statement selectively replaces characters or groups of characters in the designated field.

The REPLACING phrase names a search argument consisting of a character string of one or more characters and a condition under which the string may be applied to the field being inspected. Associated with the search argument is the replacement value, which must be the same length as the search argument. Each time the search argument finds a match in the field being inspected, under the condition stated, the replacement value replaces the matched characters.

A BEFORE/AFTER phrase may be used to delimit the area of the field being inspected. A search argument applies only to the delimited area of the field.

3.9.6.1 The Search Argument - The search argument of the REPLACING phrase names a character string and a condition under which the character string should be compared to the delimited string being inspected. Figure 3-66 shows the format of the search argument:

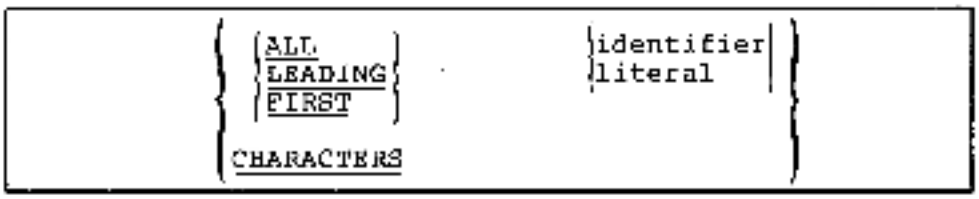


Figure 3-66
Format of the Search Argument

The CHARACTERS form of the search argument specifies that every character in the delimited string being inspected should be considered to match an imaginary character that serves as the search argument. Thus, the replacement value replaces each character in the delimited string. (The replacement value, in this case, must be one character long.)

The ALL, LEADING, and FIRST forms of the search argument specify a particular character string, which may be represented by a literal or an identifier. The search argument character string may be any length. However, each character of the argument must match a character in the delimited string before the software considers the argument matched.

- A literal character string must be either non-numeric or a figurative constant (other than ALL literal). A figurative constant, such as SPACE, ZERO, etc., represents a single character and can be written as " ", "0", etc. with the same effect. Since a figurative constant represents a single character, the replacement value must be one character long.

- An identifier must represent an elementary item of DISPLAY usage. It may be any class. However, if it is other than alphabetic, the software performs an implicit redefinition of the item. (This redefinition is identical to the BEFORE/AFTER delimiter redefinition discussed in Section 3.9.1.)

The words ALL, LEADING, and FIRST supply conditions which further delimit the inspection operation:

- The word ALL specifies that each match that the search argument finds in the delimited string is to be replaced by the replacement value. When a literal follows the word ALL, it does not have the same meaning as the figurative constant, ALL literal. (The figurative constant meaning of ALL "," is a string of consecutive commas, as many as the context of the statement requires.) ALL "," as a search argument of the REPLACING phrase means, "replace each comma without regard to adjacent characters."

- The word LEADING specifies that only adjacent matches of the search argument at the left-hand end of the delimited character string be replaced. At the first failure to match the search argument, the software terminates the replacement operation and causes the argument to become inactive.

- The word FIRST specifies that only the leftmost character string that matches the search argument is to be replaced. After the replacement operation, the search argument containing this condition becomes inactive.

3.9.6.2 The Replacement Value - Whenever the search argument finds a match in the field being inspected, the matched characters are replaced by the replacement value. The word BY followed by an identifier or literal specifies the replacement value.

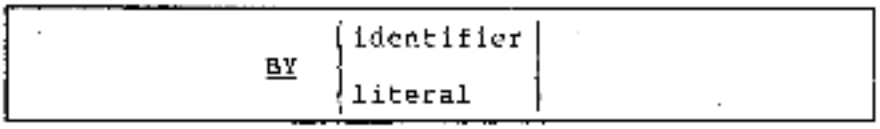


Figure 3-67
Format of the Replacement Value

The replacement value must always be the same size as its associated search argument.

If the replacement value is a literal character string, it must be either a non-numeric literal or a figurative constant (other than ALL literal). A figurative constant represents as many characters as the length that the search argument requires.

If the replacement value is an identifier, it must be an elementary item of DISPLAY usage. It may be any class. However, if it is other than alphanumeric, the software conducts an implicit redefinition of the item. (This redefinition is the same as the BEFORE/AFTER redefinition discussed in Section 3.9.1.)

3.9.6.3 The Replacement Argument - The replacement argument consists of the search argument (with its condition and character string), the replacement value, and an optional BEFORE/AFTER phrase.

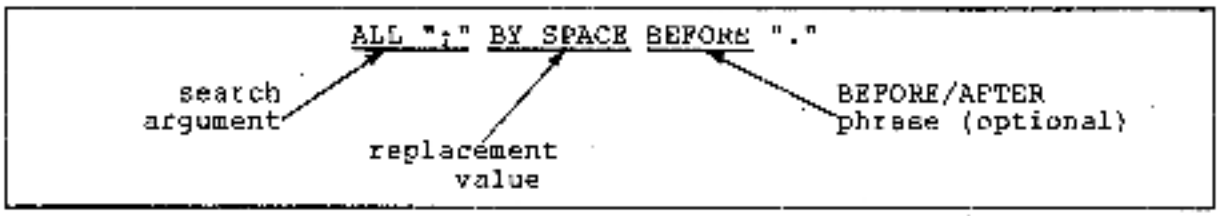


Figure 3-68
The Replacement Argument

3.9.6.4 The Replacement Argument List - One INSPECT...REPLACING statement can contain more than one replacement argument. Several replacement arguments form an argument list, and the manner in which the list is processed affects the action of any given replacement argument.

The following examples show INSPECT statements with replacement argument lists. The text following each one tells how that list will be processed.

```
INSPECT FIELD1 REPLACING
        ALL "," BY SPACE
        ALL "." BY SPACE
        ALL ";" BY SPACE.
```

Figure 3-69
Replacement Argument List that is
Active Over the Entire Field

These three replacement arguments all have the same replacement value, SPACE, and are active over the entire field being inspected.

Thus, this statement replaces all commas, periods, and semicolons with space characters; and leaves all other characters unchanged.

```
INSPECT FIELD1 REPLACING
        ALL "0" BY "1"
        ALL "1" BY "0".
```

Figure 3-70
Replacement Argument List that
"Swaps" Ones for Zeroes and Zeroes for Ones

Each of these two replacement arguments has its own replacement value, and is active over the entire field being inspected. This statement exchanges zeroes for ones and ones for zeroes, and leaves all other characters unchanged.

NOTE

When a search argument finds a match in the field being inspected, the software replaces that character string and scans to the next position beyond the replaced characters. It ignores the remaining arguments and applies the first argument in the list to the character string in the new position. Thus, it never inspects the new value that was supplied by the replacement operation. Because of this, the search arguments may have the same values as the replacement arguments with no chance of interference.

```
INSPECT FIELD1 REPLACING
        ALL "0" BY "1" BEFORE SPACE
        ALL "1" BY "0" BEFORE SPACE.
```

Figure 3-71
Replacement Argument List that
Becomes Inactive with the
Occurrence of a Space Character

This sample statement is identical to the statement in Figure 3-70, except that, here, the first occurrence of a space character in FIELD1 causes both arguments to become inactive.

```
INSPECT FIELD1 REPLACING
        ALL "0" BY "1" BEFORE SPACE
        ALL "1" BY "0" BEFORE SPACE
        CHARACTERS BY "*" BEFORE SPACE.
```

Figure 3-72
Argument List with Three Arguments
That Become Inactive with the
Occurrence of a Space

Just as in the argument list in Figure 3-71, the first space character causes all of these replacement arguments to become inactive. This argument list exchanges zeroes for ones, ones for zeroes, and asterisks for all other characters that are in the delimited area.

If the BEFORE phrase is removed from the third argument, that argument will remain active across all of FIELD1. Within the area delimited by the first space character, the third argument replaces all characters except ones and zeroes with asterisks. Beyond this area, it replaces all characters (including the space that delimited FIELD1 for the first two arguments and any zeroes and ones) with asterisks.

3.9.6.5 Interference in Replacement Argument Lists - When several search arguments that are active at the same time contain one or more identical characters, they may interfere with each other, and consequently have an effect on the replacement operation. This interference of one search argument with the matching of other search arguments is similar to the interference that occurs between tally arguments.

The action of a search argument is never affected by the BEFORE/AFTER delimiters of other arguments, since the software scans for delimiter matches before it scans for replacement operations.

The action of a search argument is never affected by the characters of any replacement value, since the scanner does not inspect the replaced characters again during execution of the INSPECT statement. Interference between search arguments, therefore, depends on the order of the arguments, the values of the arguments, and the active-inactive status of the arguments. (The discussion in Section 3.9.5.4 Interference in Tally Argument Lists, applies, generally, to replacement arguments as well.)

The following rules will help minimize interference in replacement argument lists:

1. Place search arguments with LEADING or FIRST conditions at the start of the list;

2. Place several arguments with the CHARACTERS condition at the end of the list;

3. Consider, very carefully, the order of appearance of any search arguments that contain one or more identical characters.

3.9.7 Common Errors, INSPECT Statement

The most common errors made when writing INSPECT statements are:

- Leaving the FOR out of an INSPECT...TALLYING statement.
- Using the word "WITH" instead of "BY" in the REPLACING phrase.
- Failing to initialize the tally counter.
- Omitting the word "ALL" e.g.:

```
INSPECT FIELD1 TALLYING TLY FOR SPACES.
```

CHAPTER 4

NUMERIC CHARACTER HANDLING

This chapter discusses numeric class data and the COBOL operations that can be performed on numeric data items. It is assumed that you have read Chapter 3, and that you understand the concept of COBOL data classes.

4.1 USAGES

The USAGE of a numeric class item specifies the form in which the data is stored in memory. VAX-11 COBOL-74 has four formats for numeric data storage: DISPLAY (which is equivalent to DISPLAY-6 and DISPLAY-7), COMPUTATIONAL (abbreviated COMP), and COMPUTATIONAL-3 (abbreviated COMP-3).

4.1.1 DISPLAY

Items with DISPLAY usage are stored as strings of characters (bytes) in decimal radix with an assumed decimal point and optional sign.

4.1.2 COMPUTATIONAL

COMPUTATIONAL usage is the standard VAX-11 binary format. A COMP item is stored as a binary value with an assumed decimal scaling position; it is automatically SYNCHRONIZED on a word boundary and stored in memory (in one, two, or four words) as follows:

PICTURE RANGE	STORAGE
S(9) to S9(4)	1 word (2 bytes)
S9(5) to S9(9)	1 longword (4 bytes)
S9(10) to S9(18)	1 quadword (8 bytes)

Figure 4-1 indicates the significance of each byte in a COMP data item by the number in parentheses. For example, "(1)" indicates that the byte contains the lowest-valued bits. Observe that the computer address (the first-referenced byte) of each COBOL data item corresponds to the low byte of the least significant word.

The number in parentheses also indicates the order of characters if the data item is redefined as an alphanumeric item. Consider an example of a two-word COMP item:

```
01  COMP-ITEM PIC 9(9) USAGE IS COMP.
01  GROUP-ITEM REDEFINES COMP-ITEM.
    03  CHARACTER-ITEM PIC X OCCURS 4 TIMES.
```

The subscripts of CHARACTER-ITEM correspond to the numbers in parentheses in Figure 4-1.

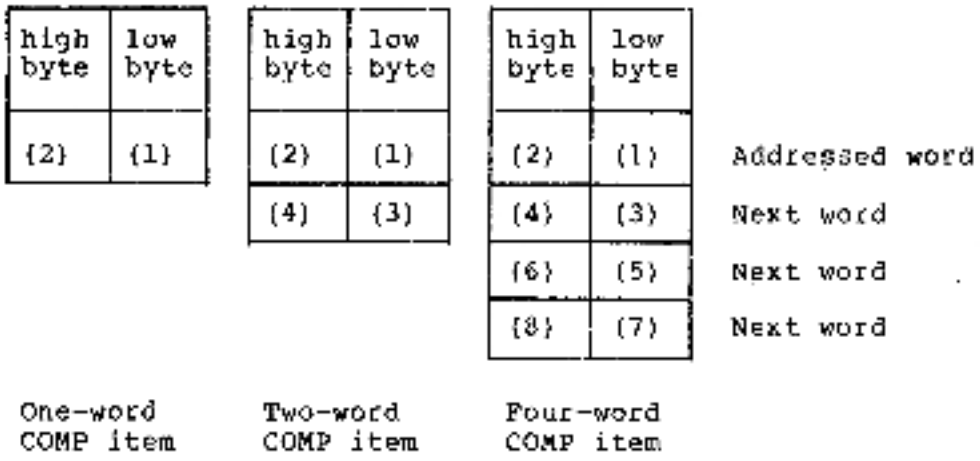


Figure 4-1
Memory Storage of COMP Data Items

4.1.3 COMPUTATIONAL-3

COMP-3 specifies packed-decimal data items. They are stored as two decimal digits per byte (byte-aligned) with an assumed decimal scaling position. The sign is contained in the rightmost half (four bits) of the rightmost byte.

The maximum size of a COMP-3 item is 18 decimal digits, regardless of the decimal scaling position. In the following example, both NUM-1 and NUM-2 represent COMP-3 items of maximum size:

```
03  NUM-1 PIC S9(18) USAGE IS COMP-3.
03  NUM-2 PIC S9(6)V9(12) USAGE IS COMP-3.
```

The description of a COMP-3 data item must have a sign in its PICTURE character-string.

4-2 NUMERIC CHARACTER HANDLING

When you specify an even number of digits, the value zero is stored in the leftmost four bits of the leftmost byte.

Signs resulting from operations in which the receiving item is specified as COMP-3 are:

"+"	binary 1100	hexadecimal C
"-"	binary 1101	hexadecimal F

The following signs are also recognized as valid, but they are not generated as a result of program operations:

Positive signs-	binary 1010,	hexadecimal A
	binary 1100,	hexadecimal C
	binary 1110,	hexadecimal E
	binary 1111,	hexadecimal F
Negative signs-	binary 1011,	hexadecimal B
	binary 1101,	hexadecimal D

Figure 4-2 represents the memory storage of COMP-3 data items of one, two, and three digits:

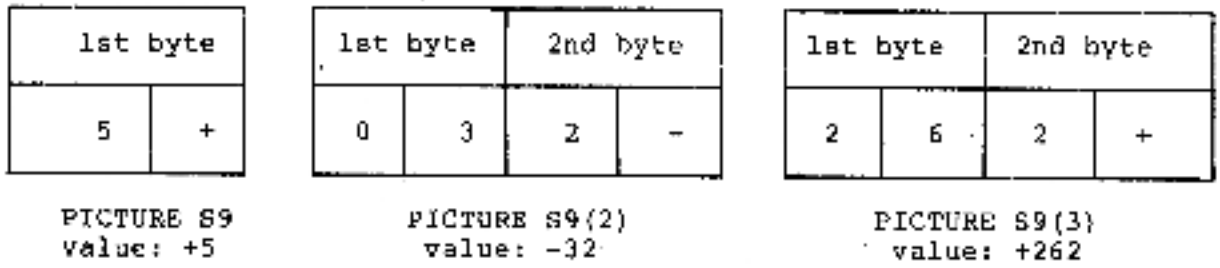


Figure 4-2
Memory Storage of COMP-3 Data Items

4.2 DECIMAL SCALING POSITION

The assumed decimal scaling position, or scaling factor, is not stored as part of an actual numeric value. However, it is used by the RTS to control operations on numeric data items. Consider the following field description:

```
01 ORDER-PRICE PIC 99V99 COMP VALUE 12.34.
```

VAX-11 COBOL-74 stores this item as a 1-word binary number. The word contains the integer value 1234 and another location contains the scaling factor. In this example, the scaling factor records the fact that this integer has two decimal fractional positions. Thus, the COBOL RTS knows that the stored binary integer is 100 times larger than the programmer intends it to be.

If the compiler encounters the following statement:

```
ADD 1 TO ORDER-PRICE.
```

it generates instructions to add a 1 to the 1234 in ORDER-PRICE. The RTS, however, scales the literal 1 up by two decimal places and adds the resultant literal, 100, to the number in ORDER-PRICE. Thus, after the ADD operation, ORDER-PRICE contains the new value 1334 (which is actually 13.34 with the stored decimal scaling position).

Thus, the VAX-11 COBOL-74 compiler and RTS manipulate the data in DISPLAY, COMP, and COMP-3 data items in much the same way. All four usages have exactly the same accuracy and precision, and can be freely mixed in a program. To illustrate, if a DISPLAY usage number and a COMP usage number are both involved in the same arithmetic statement, the RTS converts them to a common radix with no loss of information. It also converts the result, if necessary, with no loss of significance.

The only effect of specifying a binary or packed-decimal usage is that it reduces the space required for most numbers and can speed up the execution of arithmetic statements.

4.3 SIGN CONVENTIONS

COMP-3 data items must be signed; however, DISPLAY AND COMP numeric items can be signed or unsigned. Unsigned numbers can contain values that range from zero to the largest positive value allowed by their declared precision. Negative values are not allowed. All VAX-11 COBOL-74 arithmetic operations yield signed results. When the RTS must store such a result, whether positive or negative, in an unsigned data item, it stores only the absolute value of the result. Thus, unsigned items always contain zero or positive values.

This guide does not recommend unsigned numbers for general use. They are usually a source of programming errors, and are handled less efficiently than signed quantities by the RTS.

Signed quantities always contain a numeric value and an operational sign. The RTS stores the sign with the numeric value in a variety of ways depending on the usage of the item and the presence of the SIGN clause.

NOTE

If numeric data is read into a field described using the picture character S, then that data must include an operational sign of the appropriate format to pass the NUMERIC test.

VAX-11 COBOL-74 always stores signed COMP items in two's complement binary form. Thus, the high-order bit indicates the sign of the item. Sign representation for COMP-3 data items is described in Section 4.1.4.

VAX-11 COBOL-74 always stores signed DISPLAY items as a sequence of byte positions containing numeric ASCII characters. It may include the sign in the high-order byte, the low-order byte, or as a separate, extra, byte on either the high-order or low-order end of the item.

When the RTS stores the sign as part of a byte that also contains a numeric digit, the sign causes a value change in that byte and, hence, changes the value of the numeric digit. Table 4-1 shows the actual ASCII character that results when a numeric value and a sign share the same byte.

Table 4-1
The Resulting ASCII Character From a
Sign and Digit Sharing the Same Byte

		DIGIT VALUE									
		0	1	2	3	4	5	6	7	8	9
SIGN	+	{	A	B	C	D	E	F	G	H	I
	-	}	J	K	L	M	N	O	P	Q	R

A byte containing a +0 stores as hexadecimal 7B, which prints as either a { or a ; depending on the printing device.

A byte containing a -0 stores as hexadecimal 7D, which prints as either a } or a] depending on the printing device.

When the RTS stores the sign as a separate distinct character, the actual ASCII character that it stores is the graphic plus sign (hex 2B) or the graphic minus sign (hex 2D).

4.4 ILLEGAL VALUES IN NUMERIC FIELDS

All VAX-11 COBOL-74 arithmetic operations store legal values in their result fields. However, it is possible, by reading invalid data or through redefinition and group moves, to store data in numeric fields that do not obey the descriptions of those fields. (For example, it is possible to place signed values into unsigned fields, and to place non-numeric or improperly signed data into signed numeric DISPLAY fields.)

The results of arithmetic operations that use invalid data in numeric fields are unpredictable.

4.5 TESTING NUMERIC FIELDS

COBOL provides the following three kinds of tests for evaluating numeric fields:

1. Relation tests, that compare the field's contents to another numeric value;

2. Sign tests, that examine the field's sign to see if it is positive or negative; and,

3. Class tests, that inspect the field's digit positions for legal numeric values.

The following sub-sections explain these tests in detail.

4.5.1 Relation Tests

A relation test compares two numeric quantities and determines if the specified relation between them is true. For example, the following statement compares FIELD1 to FIELD2 and determines if the numeric value of FIELD1 is greater than the numeric value of FIELD2. If so, the relation condition is true and program control takes the True path of the statement.

```
IF FIELD1 > FIELD2 ...
```

Either field in a relation test may be a numeric literal or the figurative constant, ZERO. (The numeric literals 0, 00, 0.0, or ZERO are all equivalent, both in meaning and in execution speed.)

The sizes of the fields in a numeric relation test do not have to be the same (this includes the sizes of numeric literals). The comparison operation aligns both fields on their assumed decimal positions (through actual scaling operations in temporary locations or by accessing the individual digits) and supplies leading or trailing (as required) zeroes to either or both fields.

The comparison operation always compares the signs of non-zero fields and considers positive fields to be greater than negative fields. However, since it does not compare them, positive zeroes and negative zeroes are equal. (A negative zero could arrive in a field through redefinition of the field or a MOVE to a group item.) Further, the operation considers unsigned numeric fields to be positive.

The form of representation of the number (COMP, COMP-3, or DISPLAY usage) and the various methods of storing DISPLAY usage signs have no effect on numeric relation tests.

For comparison purposes, the operation converts any illegal characters stored in DISPLAY usage fields to zeroes. It does not, however, alter the actual values in those fields.

4.5.2 Sign Tests

The sign test compares a numeric quantity to zero and determines if it is greater (positive), less (negative), or equal (zero). Both the relation test and the sign test can perform this function. For example, consider the following relation test:

```
IF FIELD1 > 0 ...
```

Now consider the following sign test:

```
IF FIELD1 POSITIVE ...
```

Both of these tests accomplish the same thing and would always arrive at the same result. The sign test, however, shortens the statement and shows, at a glance, that it is testing the sign.

Table 4-2 shows the sign tests and their equivalent relation tests as applied to FIELD1.

Table 4-2
The Sign Tests

SIGN TEST	EQUIVALENT RELATION TEST
IF FIELD1 POSITIVE ...	IF FIELD1 > 0 ...
IF FIELD1 NOT POSITIVE ...	IF FIELD1 NOT > 0 ...
IF FIELD1 NEGATIVE ...	IF FIELD1 < 0 ...
IF FIELD1 NOT NEGATIVE ...	IF FIELD1 NOT < 0 ...
IF FIELD1 ZERO ...	IF FIELD1 = 0 ...
IF FIELD1 NOT ZERO ...	IF FIELD1 NOT = 0 ...

Sign tests have no execution speed advantage over relation tests. The compiler actually substitutes the equivalent relation test for every correctly written sign test. (Sections 4.2.1 and 4.2.2 discuss the acceptable sign values and the treatment of illegal sign values.)

4.5.3 Class Tests

The class test interrogates a numeric field to determine if it contains numeric or alphabetic data, and uses the result to alter the flow of control in a program. For example, the following statement determines if FIELD1 contains numeric data. If so, the test condition is true and program control takes the true path of the statement:

```
IF FIELD1 IS NUMERIC ...
```

When reading in newly prepared data, it is often desirable to check certain fields for valid values. Relation tests and sign tests can only determine if the field's contents are within a certain range, and these tests both treat illegal characters in DISPLAY usage items as zeroes. Thus, some data preparation errors could pass both of these tests.

The NUMERIC class test checks numeric (or alphanumeric) DISPLAY usage fields for valid numeric digits.

If the field being tested contains a sign (whether carried as an overpunch or as a separate character), the test checks it for a valid sign value. If the character position carrying the sign contains an illegal sign value, the NUMERIC class test rejects the item and program control takes the false path of the IF statement. If the character position contains a valid sign and all digit positions in the field contain valid numeric digits, the NUMERIC class test passes the item and program control takes the true path of the IF statement.

The ALPHABETIC class test checks alphabetic (or alphanumeric) fields for valid alphabetic characters and the space character. If all of the character positions of the field contain ASCII characters (A-Z or space), the item passes the ALPHABETIC class test and causes program control to take the true path of the IF statement. (For further information concerning the ALPHABETIC class test, see Chapter 3, Section 3.3.2.)

4.6 THE MOVE STATEMENT

The MOVE statement moves the contents of one field into another. The following sample MOVE statement moves the contents of FIELD1 into FIELD2.

```
MOVE FIELD1 TO FIELD2.
```

Section 3.5 discusses the basic MOVE statement. This section considers MOVE statements as applied to numeric fields. These MOVE statements can be grouped into the following three categories:

1. Group moves,
2. Elementary moves with numeric receiving fields, and
3. Elementary moves with numeric edited receiving fields.

The following three sub-sections (4.4.1, 4.4.2, and 4.4.3) discuss each of these categories separately.

4.6.1 Group Moves

The software considers a move to be a group move if either the sending field or the receiving field is a group item. It treats both fields in a group move as alphanumeric class fields and performs the move as an alphanumeric to alphanumeric elementary move.

If either field in a group move is a numeric elementary item, the RTS treats the storage area occupied by that item as a field of alphanumeric bytes; thus, it ignores the USAGE, sign, and decimal point location characteristics of the numeric item.

Only the item's allocated size, in bytes, affects the move operation. The RTS considers a separate sign character to be part of the item and moves it with the numeric digit positions.

4.6.2 Elementary Numeric Moves

If both fields of a MOVE statement are elementary items and the receiving field is numeric, the RTS considers the move to be an elementary numeric move. (The sending field may be either numeric or alphanumeric.) The numeric receiving field may be DISPLAY, COMP, or COMP-3 usage. The elementary numeric move converts the data format of the sending field to the data format of the receiving field.

An alphanumeric sending field may be either an elementary data item or any alphanumeric literal other than the figurative constants SPACE, QUOTE, LOW-VALUE, HIGH-VALUE, or ALL "literal". The elementary numeric move accepts the figurative constant ZERO and considers it to be equivalent to the numeric literal 0. It treats alphanumeric sending fields as unsigned integers of DISPLAY usage.

If necessary, the numeric move operation converts the sending field to the data format of the receiving field and aligns the sending field's decimal point on that of the receiving field. It then moves the sending field digits to their corresponding receiving field digits.

If the sending field has more digit positions than the receiving field, the decimal point alignment operation truncates the sending field, with the resultant loss of digits. The end truncated (high-order or low-order) depends upon the number of sending field digit positions that find matches on each side of the receiving field's decimal point. If the receiving field has fewer digit positions on both sides of the decimal point, the operation truncates both ends of the sending field. Thus, if a field described as PIC 999V999 is moved to a field described as PIC 99V99, it loses one digit from the left end and one from the right end. Figure 4-3 illustrates this alignment operation (the caret (^) indicates the stored decimal scaling position):

```
        01 AMOUNT1 PIC 99V99.

           ...

           MOVE 123.321 TO AMOUNT1.

    Before execution   00 00

    After execution    23 32
```

Figure 4-3
Truncation Caused By Decimal Point Alignment

If the sending field has fewer digit positions than the receiving field, the move operation supplies zeroes for all unfilled digit positions. Figure 4-4 illustrates this alignment (the carat (^) indicates the stored decimal scaling position):

```
        01 TOTAL-AMT PIC 999V99.

           ...

           MOVE 1 TO TOTAL-AMT.
```

Before execution	000 00
After execution	001 00

Figure 4-4
Zero Filling Caused By Decimal Point Alignment

The following statement produces the same results:

```
MOVE 001.00 TO TOTAL-AMT.
```

Consider the following two MOVE statements and their resultant truncating and zero-filling effects:

STATEMENT	TOTAL-AMT AFTER EXECUTION
MOVE 00100 TO TOTAL-AMT	100 00
MOVE "00100" TO TOTAL-AMT	100 00

Literals with leading or trailing zeroes have no significant advantage in space or execution speed with VAX-11 COBOL-74, and the zeroes are often lost by decimal point alignment.

The MOVE statement's receiving field dictates how the sign will be moved. A signed DISPLAY usage receiving field causes the sign to be moved as a separate quantity. An unsigned DISPLAY usage receiving field causes no sign movement. A COMP usage receiving field, whether signed or unsigned, causes the sign to be moved; however, if the receiving field is unsigned, the RTS sets its value to absolute. A COMP-3 receiving field always causes the sign to be moved.

4.6.3 Elementary Numeric Edited Moves

The VAX-11 COBOL-74 run-time system considers an elementary numeric move to a receiving field of the numeric edited category to be an elementary numeric edited move. The sending field of an elementary numeric edited move may be either numeric or alphanumeric and, if numeric, its usage can be DISPLAY, COMP, or COMP-3. The RTS treats alphanumeric sending fields in numeric edited moves as unsigned DISPLAY usage integers.

The RTS considers the receiving field to be numeric edited category if it is described with a BLANK WHEN ZERO clause, or a combination of the following symbols:

B	Space insertion position;
P	Decimal scaling position;
V	Location of assumed decimal point;
Z	Leading numeric character position to be replaced by a space if the position contains a zero;
0	Zero insertion position;
9	Position contains a numeric character;
/	Slash insertion position;
,	Comma insertion position;
.	Decimal point insertion position;
*	Leading numeric character position to be replaced by an asterisk if the position contains a zero;
+	Positive editing sign control symbol;
-	Negative editing sign control symbol;
CR	Credit editing sign control symbol;
DB	Debit editing sign control symbol;
cs	Currency symbol ($) insertion position.

A numeric edited field may contain 9, V, and P, but combinations of those symbols without an editing character do not make the field numeric edited.

The numeric edited move operation first converts the sending field to DISPLAY usage and aligns both fields on their decimal point locations, truncating or padding (with zeroes) the sending field until it contains the same number of digit positions on both sides of the decimal point as the receiving field. It then moves the resulting digit values to the receiving field digit positions following the COBOL editing rules.

The COBOL editing rules allow the numeric edited move operation to perform any of the following editing functions:

- Suppress leading zeroes with either spaces or asterisks;

- Float a currency sign and a plus or minus sign through suppressed zeroes, inserting the sign at either end of the field;
- Insert zeroes and spaces;
- Insert commas and a decimal point.

Figure 4-5 illustrates several of these functions with the statement, MOVE FLD-B TO TOTAL-AMT. (Assume that FLD-B is described as S9999V99.)

FLD-B	TOTAL-AMT	
	PICTURE STRING	CONTENTS AFTER MOVE
0023 00	ZZZZ.99	23.00
0085 90	++++.99	-85.96
1234 00	Z,ZZZ.99	1,234.00
0012 34	$,$$$.99	$12.34
0000 34	$,$89.99	$0.34
1234 00	$$,$$$.99	$1,234.00
0012 34	$$9,999.99	$0,012.34
0012 34	$$$$,$$$.99	$12.34
0000 00	$$$,$$$.$$	
0012 3M	++I+.99	-12.34
0012 34	$***,***.99	$*****12.34

Figure 4-5
Numeric Editing

The currency symbol ($) and the editing sign control symbols (+ -) are the only floating symbols. To float them, enter a string of two or more occurrences of the symbol.

4.6.4 Common Errors, Numeric MOVE Statements

The most common errors made when writing numeric MOVE statements are:

- Placing an incorrect number of replacement characters in a numeric edited item.
- Moving non-numeric data into numeric fields with group moves.
- Trying to float the $ or + insertion characters past the decimal point to force zero values to appear as .00 instead of spaces. (Use $$.99 or ++.99.)
- Forgetting that the $ or + insertion characters require an additional position on the leftmost end that cannot be replaced by a digit (unlike the * insertion character which can be completely replaced).

4.7 THE ARITHMETIC STATEMENTS

The COBOL arithmetic statements, ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE allow COBOL programs to perform simple arithmetic operations on numeric data.

This section covers the use of the COBOL arithmetic statements. The first five sub-sections (4.7.1 through 4.7.5) discuss the common features of the statements and the last five (4.7.6 through 4.7.10) discuss the individual arithmetic statements themselves.

4.7.1 Intermediate Results

Most forms of the arithmetic statements perform their operations in temporary work locations, then move the results to the receiving fields, aligning the decimal points and truncating or zero filling the resultant values.

This temporary work field, called the intermediate result field, has a maximum size of 18 numeric digits. The actual size of the intermediate result field varies for each statement, and is determined at compile time based on the sizes of the operands used by the statement.

When the compiler determines that the size of the intermediate result field exceeds 18 digits, it truncates the excess high-order digits. Thus, a program that requests a multiplication operation between the following two fields,

```
PIC 9(18) and PIC V99.
```

(which would otherwise cause the compiler to set up a 20-digit intermediate result field -- 9(18)V99) actually causes the following intermediate result field

```
PIC 9(16)V99.
```

VAX-11 COBOL-74 truncates high-order digits or low-order digits to the right of the decimal point, based on the assumption that most large data declarations are larger than ever need be, so zeroes occupy most of their high-order digit positions. Numeric data may be declared as PIC 9(12) or PIC 9(15) but the values that are placed in these fields will probably not exceed nine digits of range (1 billion) in most applications.

When using large numbers (or numbers with many decimal places) that are close to 18 digits long, examine all of the arithmetic operations that manipulate those numbers to determine if truncation will occur.

If truncation is a possibility, reduce the size of the number by dividing it by a power of 10 prior to the arithmetic operation. (This scaling down operation causes the low-order end to lose digits, but these are probably less critical.) Then, after the arithmetic operation, multiply the result by the same power of 10.

To save the low-order digits in such an operation, move the field to a temporary location before the scaling DIVIDE, perform separate, identical arithmetic operations on both the original and the temporary fields, then, after the scaling MULTIPLY, combine their results.

4.7.2 The ROUNDED Phrase

Rounding-off is an important tool with most arithmetic operations. The ROUNDED phrase causes the RTS to round-off the results of COBOL arithmetic operations.

The phrase may be used on any COBOL arithmetic statement. Rounding-off takes place only when the ROUNDED phrase requests it, and then only if the intermediate result has more low-order digits than the result field.

VAX-11 COBOL-74 rounds-off by adding a 5 to the leftmost truncated digit of the absolute value of the intermediate result before it stores that result.

Consider the following illustration and assume an intermediate result of 54321.2468:

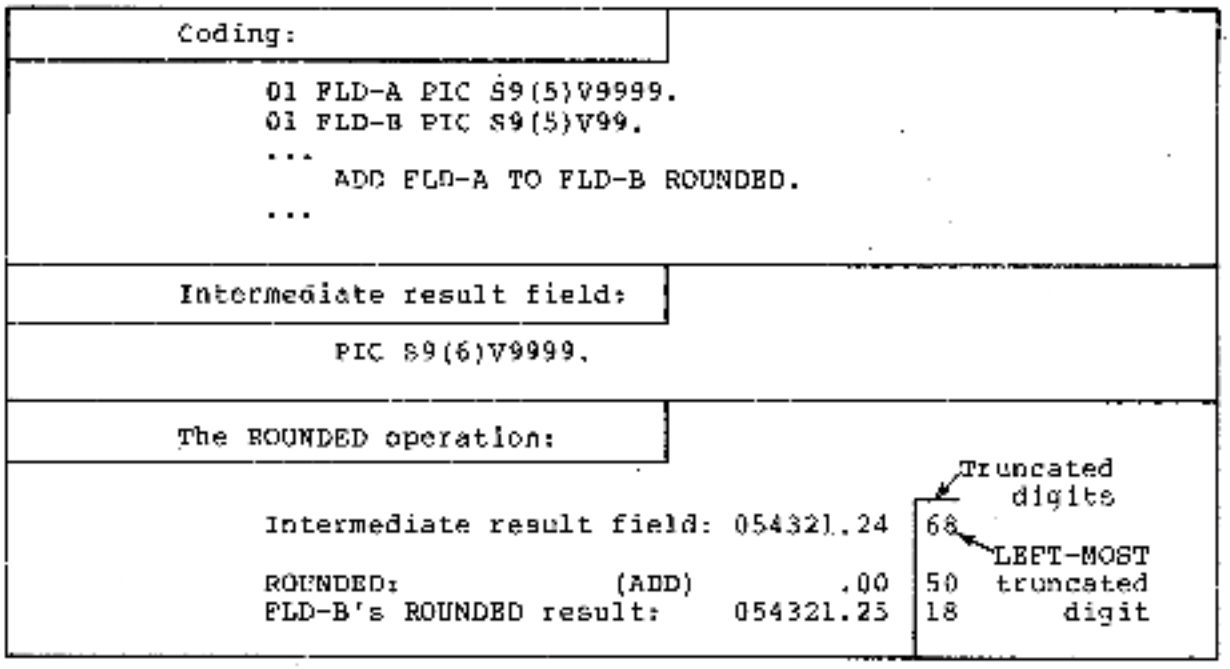


Figure 4-6
Rounding Truncated Decimal Point Positions

The following ROUNDING example rounds-off to the decimal scaling position (P). Assume an intermediate result of 24680. (Section 4.7.4 discusses the GIVING phrase in numeric operations.)

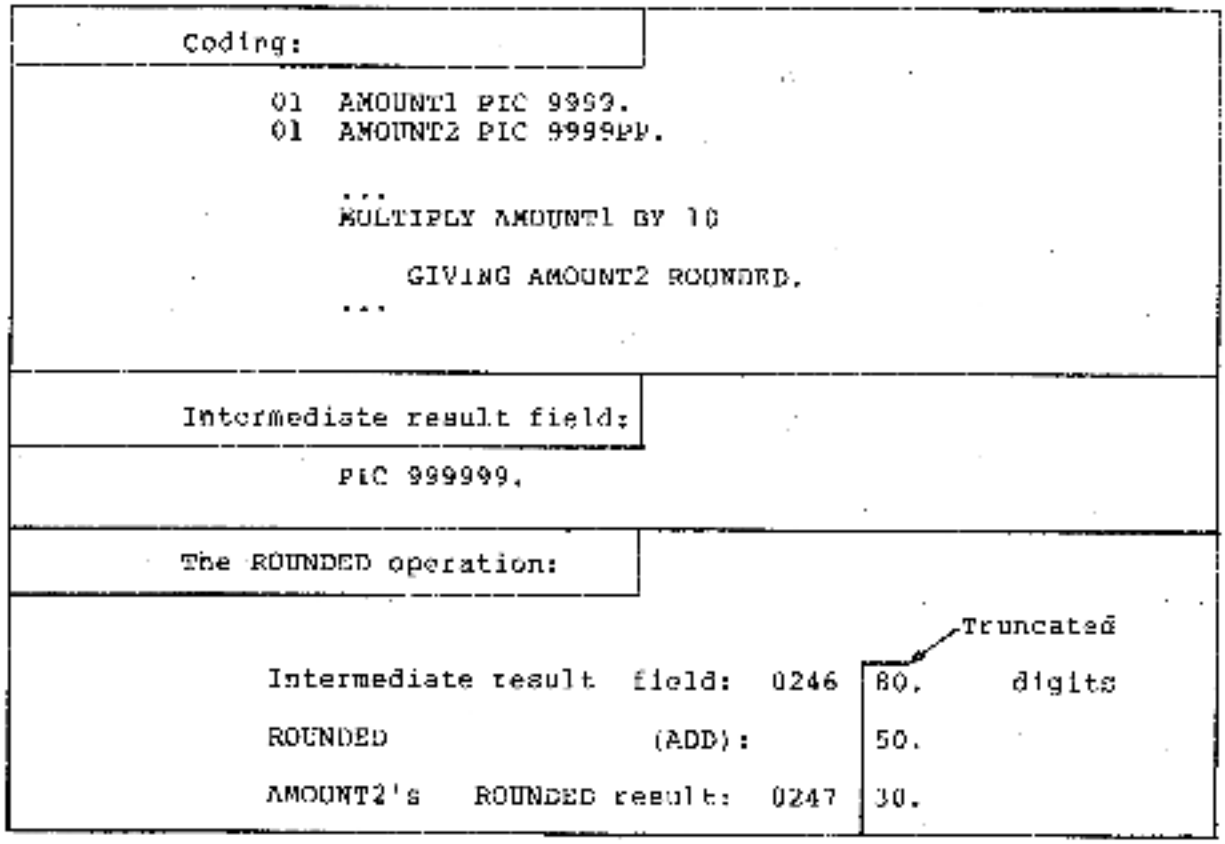


Figure 4-7
Rounding Truncated Decimal Scaling Positions

4.7.3 The SIZE ERROR Phrase

The SIZE ERROR phrase detects the loss of high-order non-zero digits in the results of COBOL arithmetic operations.

The phrase may be used on any COBOL arithmetic statement.

When the execution of a statement with no SIZE ERROR phrase results in a size error, the RTS truncates the high-order digits and stores the result without notifying the user. When the execution of a statement with a SIZE ERROR phrase results in a size error, the RTS discards the entire result (it does not alter the receiving fields in any way) and executes the SIZE-ERROR imperative phrase.

If the statement contains both ROUNDED and SIZE ERROR phrases, the RTS rounds the result before it checks for a size error.

The phrase cannot be used on numeric MOVE statements. Thus, if a program moves a numeric quantity to a smaller numeric field, it may inadvertently lose high-order digits. For example, consider the following MOVE of a field to a smaller field:

```
01 AMOUNT-A PIC 9(8)V99.

01 AMOUNT-B PIC 9(4)V99.

   ...

MOVE AMOUNT-A TO AMOUNT-B.
```

This MOVE operation always loses four of AMOUNT-A's high-order digits. Either of the following two statements could determine whether these digits are zero or non-zero, and could be tailored to any size field:

```
1.  IF AMOUNT-A NOT > 9999.99
    MOVE AMOUNT-A TO AMOUNT-B
    ELSE ...

2.  ADD ZERO TO AMOUNT-A GIVING AMOUNT-B
    ON SIZE ERROR ...
```

Both of these alternatives allow the MOVE operation to occur only if AMOUNT-A loses no significant digits. If the value in AMOUNT-A is too large, both alternatives avoid altering AMOUNT-B and take the alternative execution path.

4.7.4 The GIVING Phrase

The GIVING phrase moves the intermediate result field of an arithmetic operation to a receiving field. (The phrase acts exactly like a MOVE statement with the intermediate result serving as a sending field and the data item following the word GIVING (in the statement) serving as a receiving field.)

The phrase may be used on the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements.

If the data item following the word GIVING is a numeric edited field, the RTS performs the editing the same way it does for MOVE statements.

4.7.5 Multiple Operands in ADD and SUBTRACT Statements

Both the ADD and SUBTRACT statements may contain a string of more than one operand preceding the word TO, FROM, or GIVING.

Multiple operands in either of these statements cause the RTS to add the string of operands together and use the intermediate result of that operation as a single operand to be added to or subtracted from, the receiving field.

The following three equivalent coding groups illustrate how the software executes the multiple operand statements:

1. Statement:
```
ADD A B C D TO E F G H.
```

   Equivalent coding:
```
ADD A B, GIVING TEMP.
ADD TEMP, C, GIVING TEMP.
ADD TEMP, D, GIVING TEMP.
ADD TEMP, E, GIVING E.
ADD TEMP, F GIVING F.
ADD TEMP, G GIVING G.
ADD TEMP, H GIVING H.
```

2. Statement:
```
SUBTRACT A, B, C, FROM D.
```

   Equivalent coding:
```
ADD A, B, GIVING TEMP.
ADD TEMP, C GIVING TEMP.
SUBTRACT TEMP FROM D GIVING D.
```

3. Statement:
```
ADD A B C D GIVING E.
```

   Equivalent coding:
```
ADD A B GIVING TEMP.
ADD TEMP C GIVING TEMP.
ADD TEMP D GIVING E.
```

(Just as with all COBOL statements, any commas in these statements are optional.)

Only statement 3 may have a numeric edited receiving field, since it is the only statement containing a GIVING phrase.

4.7.6 The ADD Statement

The ADD statement adds two or more operands together and stores the result.

The statement may contain multiple operands (with the exception of Format 3) and the ROUNDED and SIZE ERROR phrases. It may be written in one of the following formats:

```
Format 1.    ADD FIELD1 ...TO FIELD2 FIELD3 ... .

Format 2.    ADD FIELD1 FIELD2 ...GIVING FIELD3 FIELD4 ... .

Format 3.    ADD CORRESPONDING FIELD1 TO FIELD2.
```

In Format 1, the receiving fields (FIELD2, FIELD3) are one of the addends. These must not be in the numeric edited category.

In Format 2, the receiving fields (FIELD3, FIELD4) are not one of the addends. They may either be numeric or numeric edited. When using this format, omit the word TO.

In Format 3, the receiving field (FIELD2) is one of the addends. Both FIELD1 and FIELD2 must be group items. The corresponding elements of FIELD1 are added to the corresponding elements of FIELD2.

4.7.7 The SUBTRACT Statement

The SUBTRACT statement subtracts one, or the sum of two or more, operands from another operand and stores the result.

The statement may contain multiple operands (with the exception of Format 3) and the ROUNDED and SIZE ERROR phrases. It may be written in one of the following formats:

```
Format 1.    SUBTRACT FIELD1 ...  FROM FIELD2 FIELD3 ... .

Format 2.    SUBTRACT FIELD1 ...  FROM FIELD2
             GIVING FIELD3 FIELD4 ... .

Format 3.    SUBTRACT CORRESPONDING FIELD1 FROM FIELD2.
```

In Format 1, the receiving fields (FIELD2, FIELD3) are both the subtrahend and the difference (the result). These must not be in the numeric edited category.

In Format 2, the receiving fields (FIELD3, FIELD4) are used only to store the result. They may be either numeric or numeric edited.

In Format 3, the receiving field (FIELD2) is both the subtrahend and the difference (results). Both FIELD1 and FIELD2 must be group items. The corresponding elements of FIELD2.

4.7.8 The MULTIPLY Statement

The MULTIPLY statement multiplies one operand by another and stores the result.

The statement may contain the ROUNDED and SIZE ERROR phrases. It may contain multiple receiving operands. It may be written in either of the following formats:

```
Format 1.    MULTIPLY FIELD1 BY FIELD2, FIELD3 ... .

Format 2.    MULTIPLY FIELD1 BY FIELD2 GIVING FIELD3, FIELD4 ... .
```

In Format 1, the receiving fields (FIELD2, FIELD3) are also the multipliers. These must not be in the numeric edited category.

In Format 2, the receiving fields (FIELD3, FIELD4) are neither multiplier nor multiplicand. These may be either numeric or numeric edited.

COBOL's "near English" format could cause a problem with the MULTIPLY statement, since it is common to speak of multiplying a number (multiplicand) by another number (multiplier) and to think of the result as a new value for the multiplicand; thus:

```
MULTIPLY EARNINGS BY 0.24.
                        Multiplier
              Multiplicand
```

This statement is incorrect since the RTS stores the result in the multiplier field, and this multiplier is a literal. The compiler could diagnose this error, but would not diagnose it if the multiplier were a data item. Consider this multiplier written as a data item:

```
MULTIPLY EARNINGS BY TAX-RATE.
```

The compiler would not diagnose this statement's error, and would store the result of the operation in TAX-RATE. A good practice when using MULTIPLY statements is to always write them in Format 2. This ensures that the result is properly stored. The following two statements safely capture their results:

```
MULTIPLY EARNINGS BY 0.24 GIVING EARNINGS.
```

or

```
MULTIPLY EARNINGS BY TAX-RATE GIVING EARNINGS.
```

4.7.9 The DIVIDE Statement

The DIVIDE statement divides one operand into another and stores the result.

The statement may contain the ROUNDED and SIZE ERROR phrases. With the exception of Formats 4 and 5, it may not contain multiple receiving operands. It may be written in any of the following formats:

```
Format 1.   DIVIDE FIELD1 INTO FIELD2 FIELD3 ... .

Format 2.   DIVIDE FIELD1 INTO FIELD2 GIVING FIELD3 FIELD4 ... .

Format 3.   DIVIDE FIELD2 BY FIELD1 GIVING FIELD3 FIELD4 ... .

Format 4.   DIVIDE FIELD1 INTO FIELD2 GIVING FIELD3 REMAINDER
            FIELD4.

Format 5.   DIVIDE FIELD1 BY FIELD2 GIVING FIELD3 REMAINDER
            FIELD4.
```

In Format 1, the receiving fields (FIELD2, FIELD3) are also the dividends. These must not be in the numeric edited category.

In Formats 2 and 3, the receiving fields (FIELD3, FIELD4 ...) are neither dividends nor divisor. These may be either numeric or numeric edited.

In Formats 4 and 5, the receiving field (FIELD3) is neither a dividend nor a divisor. FIELD4 is the remainder. The receiving field and the remainder may be either numeric or numeric edited.

4.7.10 The COMPUTE Statement

The COMPUTE statement computes the value of an arithmetic expression and stores the value in the result.

The statement may contain the ROUNDED and SIZE ERROR phrases. It may contain multiple receiving operands. The COMPUTE statement has the following format:

```
COMPUTE FIELD1 FIELD2 ... = arithmetic-expression.
```

The receiving fields (FIELD1, FIELD2) may be either numeric or numeric edited.

4.7.11 Common Errors, Arithmetic Statements

The most common errors made when using arithmetic statements are:

- Using an alphanumeric class field in an arithmetic statement. The MOVE statement allows data movement between alphanumeric class fields and certain numeric class fields, but arithmetic statements require that all fields be numeric.
- Writing the ADD or SUBTRACT statements without the GIVING phrase, but attempting to put the result into a numeric edited field.
- Writing a Format 2 ADD statement with the word TO; For example:

```
ADD A TO B GIVING C.
```

- Subtracting a 1 from a numeric counter that was described as an unsigned quantity, and testing for a value of less than zero.
- Forgetting that the MULTIPLY statement, without the GIVING phrase, stores the result back into the second operand (multiplier).

- Performing a series of calculations in such a way as to generate an intermediate result that is larger than 18 digits when the final result will be fewer digits. (The programmer should be careful to intersperse divisions with multiplications or to drop non-significant digits that result from multiplying large numbers (or numbers with many decimal places).

- Performing an operation on a field that contains a value greater than the precision of its data description. This can happen only if the field was disarranged by a group move or redefinition.

- Forgetting that, in an arithmetic statment containing multiple receiving fields, the ROUNDED phrase must be specified for each receiving field that is to be rounded.

- Forgetting that, in an arithmetic statement containing multiple receiving fields, the ON SIZE ERROR phrase, if specified, applies to all receiving fields. Only those receiving operands for which a size error condition is raised are left unaltered. The ON SIZE ERROR statement is executed after all the receiving fields are processed by the RTS.

4.8 ARITHMETIC EXPRESSION PROCESSING

COBOL provides language facilities for manipulating user-defined data arithmetically. In particular, the language provides the arithmetic statements ADD, SUBTRACT, MULTIPLY, and DIVIDE and the facilities of arithmetic expressions using the +, -, *, /, and ** operators. In simple terms, a given arithmetic functionality may be expressed in one of several ways. For example, consider a COBOL application in which the total yearly sales of a saleeman are to be computed as the sum of the four individual sales quarters. Figure 4-8 illustrates one method of expressing a solution to this problem in COBOL:

```
MOVE 0 TO TEMP.
ADD 1ST-SALES TO TEMP.
ADD 2ND-SALES TO TEMP.
ADD 3RD-SALES TO TEMP.
ADD 4TH-SALES TO TEMP GIVING TOTAL-SALES.
```

Figure 4-8 Explicit Programmer-Defined Temporary Work Area

In figure 4-8, the COBOL programmer chooses to use a series of single ADD statements to develop the final value for TOTAL-SALES. In the process of computing TOTAL-SALES, a COBOL data-name, called TEMP, is used to develop the partial sums (i.e., intermediate results). The important point here is that the programmer explicitly defines and declares the temporary work area TEMP in the data division of the COBOL program. That is, the attributes (i.e., class, USAGE, number of integer and decimal places to be maintained) are specified explicitly by the COBOL programmer.

Figure 4-9 below illustrates another way of expressing a solution to the problem:

```
ADD 1ST-SALES, 2ND-SALES, 3RD-SALES, 4TH-SALES
    GIVING TOTAL-SALES.
```

FIGURE 4-9
ARITHMETIC STATEMENT INTERMEDIATE RESULT FIELD ATTRIBUTES
DETERMINED FROM COMPOSITE OF OPERANDS

IN THIS EXAMPLE, THE PROGRAMMER CHOOSES TO COMPUTE TOTAL-SALES with a single ADD statement. Analogous to the previous example, an intermediate result field is required to develop the partial sums of the four quarterly sales quantities. In Figure 4-8, the programmer is cognizant of this requirement, but chose to define the intermediate result area TEMP explicitly in the data division of his COBOL program. However, for the example in Figure 4-9, the compiler defines the intermediate result field in a manner transparent to the COBOL source program. That is, the compiler allocates storage for and assigns various attributes to this "transparent" intermediate result field according to a well-defined set of rules defined by the COBOL language specification. In particular, the attributes of number-of-integer-places, number-of-decimal-places, and USAGE assigned by the software to the intermediate result field are a function of the composite of source operands in the ADD statement. (The reader should read the **VAX-11 COBOL-74 Reference Manual** for details concerning the composite of operands for the arithmetic statements.) The important point here is that the ANS-74 COBOL language standard prescribes rules for determining the attributes of intermediate result fields for the arithmetic statements, and the language processor, the VAX-11 COBOL-74 compiler, must implement those rules.

As a final example, consider the following solution to our problem:

```
COMPUTE TOTAL-SALES = 1ST-SALES + 2ND-SALES + 3RD-SALES
        + 4TH-SALES.
```

Figure 4-10
Arithmetic Expression Intermediate Result Field
Attributes Determined by Implementor-Defined Rules

In Figure 4-10, the programmer solves the problem by using a single COMPUTE statement with an embedded arithmetic expression. Again, an intermediate result field is required and, as in Figure 4-9, is defined by the software. However, in defining the attributes of intermediate result fields for COBOL arithmetic expressions, the ANS-74 COBOL language standard is not as helpful to the user as it could be. In fact, the COBOL language standard gives almost complete freedom to the implementor in defining the attributes of the arithmetic expression intermediate result fields. The only rules imposed by the ANS-74 COBOL language specifications are:

1. Arithmetic operations are to be combined without restrictions on the composite of operands and/or receiving fields.

2. Each implementor will indicate techniques used in handling arithmetic expressions.

Thus, the user can and should expect differences between various implementations of ANS-74 COBOL. The rest of this section describes how the VAX-11 COBOL-74 compiler computes the sizes of intermediate result fields.

The compiler computes the size of an intermediate result field for each component operation of an arithmetic expression. Each operation can be stated as:

```
OP1  OPR  OP2
```

where:

OP1 is the first operand

OPR is an arithmetic operator

OP2 is the second operand

The size of an intermediate result is described in terms of the number of integer places (IP) and the number of decimal places (DP). The symbol DPEXP represents the maximum number of decimal places in the entire arithmetic expression.

```
  OPR

+ and -       IP = max(IP(OP1), IP(OP2)) + 1
              DP = max(DP(OP1), DP(OP2))

  *           IP = IP(OP1) + IP(OP2)
              DP = DP(OP1) + DP(OP2)

  /           IP = IP(OP1) + DP(OP2)
              DP = max(DPEXP, max(DP(OP1), DP(OP2) + 1))

  **    For exponents that convert to one-word values,
                a = OP2
                b = OP2 + DP(OP1)

        Otherwise,

                a = 9, if IP(OP2) = 1,
                    otherwise, a = 19
                b = DPEXP

        and

             IP = IP(OP1) * a
             DP = max(DPEXP, DP(OP1) * b)
```

CHAPTER 5

TABLE HANDLING

5.1 INTRODUCTION

With COBOL, as with any other language, any data item to which the program refers must be uniquely identified. This unique identification of data items is usually accomplished by assigning a unique name to each item. However, in many applications this is tedious and inconvenient; often programs require too many names for items that have different names but contain the same type of information. Tables provide a simple solution to this problem. VAX-11 COBOL-74 includes full table handling capabilities as outlined for standard COBOL in the 1974 ANSI Standards.

A table is a repetition of one item (element) in memory. This repetition is accomplished by the use of the OCCURS clause in the data description entry. The literal value in the OCCURS clause causes the software to duplicate the data description entry as many times as indicated by that value, thus creating a matrix or table.

The elements may be initialized with the VALUE clause or with a procedural instruction. They may contain synchronized or unsynchronized data. They may be accessed only with subscripted procedural instructions. A subscript is a parenthesized integer or data name (with an integer value). The integer value represents the desired occurrence of the element.

This chapter discusses how to set up tables and access them accurately and efficiently. It attempts to cover any problems that may be encountered while handling tables. Read it through carefully before setting up tables with VAX-11 COBOL-74.

5.2 DEFINING TABLES

To define a table with VAX-11 COBOL-74, simply complete a standard data description for one element of the table and follow it with an OCCURS clause. The OCCURS clause contains an integer which dictates the number of times that element will be repeated in memory, thus creating a table.

The OCCURS phrase has two formats:

```
Format 1

OCCURS integer-2 TIMES

[ {ASCENDING } KEY IS data-name-2  [, data-name-3] ...] ...
  {DESCENDING}

      [ INDEXED BY index-name-1  [, index-name-2] ...]
```

```
Format 2

OCCURS integer-1 TO integer-2 TIMES DEPENDING ON data-name-1

[ {ASCENDING } KEY IS data-name-2  [, data-name-3] ... ] ...
  {DESCENDING}

      [ INDEXED BY index-name-1  [, index-name-2] ...]
```

In either format, the system generates a buffer large enough to accommodate integer-2 occurrences of the data description. Therefore, the amount of storage allocated in either case is equal to the amount of storage required to repeat the data entry integer-2 times.

The software will automatically map the elements into memory. When mapping a table into memory, the software follows the rules for mapping which depend on whether the element contains synchronized items or not. If they do not contain synchronized items, the software maps them into adjacent memory locations and the size of the table can be easily calculated by multiplying the size of the element times the number of occurrences (5X10 for the table illustrated in Figure 5-1, or 50 bytes of memory).

```
01 A-TABLE
   03 A-GROUP PIC X(5) OCCURS 10 TIMES.
```

Figure 5-1
Defining a Table

5.2.1 The OCCURS Phrase - Format 1

When Format 1 is used, a fixed length table is generated, whose length (number of occurrences) is equal to the value specified by integer-2. This format is useful for storing large amounts of frequently used reference data whose size never changes. Tax tables, used in payroll deduction programs, are an excellent example of where a Format 1 (fixed length) table might be used.

5.2.2 The OCCURS Phrase - Format 2

Format 2 is used to generate variable length tables. When used, a table whose length (number of occurrences) is equal to the value specified by data-name-1 is generated.

NOTE

Data-name-1 must always be a positive integer whose value is equal to or greater than integer-1 but not greater than integer-2.

Unlike format 1 tables, the number of occurrences of data items in format 2 tables can be dynamically expanded or reduced to satisfy user needs.

By generating a variable length table, the user is, in effect, saying: "build me a table that can contain at least integer-1 occurrences, but no more than integer-2 occurrences, and set its number of occurrences equal to the value specified by data-name-1".

Data-name-1 always reflects the number of occurrences available for user access. To expand the size (number of occurrences available for use) of a table, the user need only increase the value of data-name-1 accordingly.

Likewise, reducing the value in data-name-1 will reduce the number of occurrences available for user access.

5.3 MAPPING TABLE ELEMENTS

As mentioned in Section 5.2, when the software detects an OCCURS clause in an unsynchronized item, it maps the table elements into adjacent locations in memory. Consider the following data description of a simple table and the way it is mapped into memory:

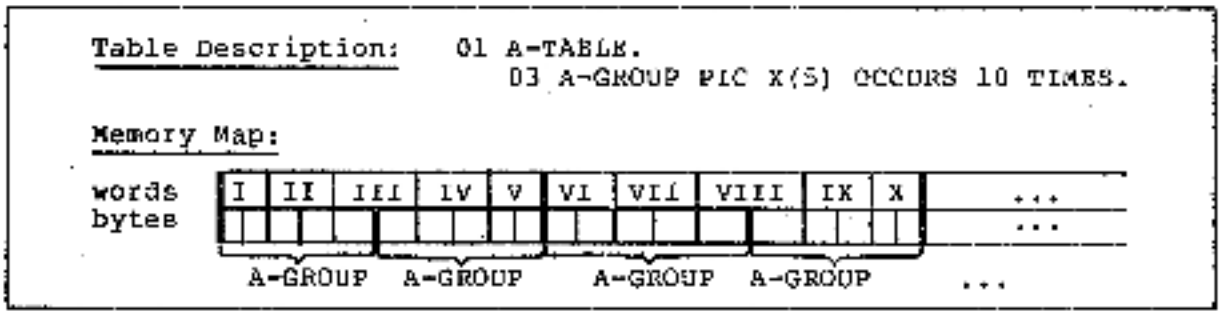


Figure 5-2
Mapping a Table into Memory

The data description in Figure 5-2 causes the software to set up ten items of five bytes each (elements) and place them in adjacent ascending memory locations for a total of 50 character positions, thus creating a table. Since the length of each A-GROUP element is odd (5), the memory addresses of each subsequent element will alternate between odd and even locations.

The SYNCHRONIZED clause causes the software to add a fill byte to items that contain an odd number of bytes, thereby making the number of bytes in that item even. This ensures that each subsequent occurrence of the element will not alternate between odd and even addresses, but will map the same (odd or even) as the first repetition of that element. If the data description of A-GROUP contained a SYNCHRONIZED clause, the software would map it quite differently. If A-GROUP were synchronized, it would expand its length to three words. The item will, by default, be synchronized to the left occupying the first five characters of the three words. The software supplies a padding character to fill out the third word. This padding character is not a part of the A-GROUP element and table instructions referring to A-TABLE will not detect the presence or absence of the character.

The padding character does, however, affect the overall length of the group item and, hence, the table. Without the SYNCHRONIZED clause, A-TABLE required only 50 character positions; now, with the clause, it requires 60 character positions. (This length includes the last padding character -- following the tenth element in the table.)

Although the SYNCHRONIZED clause has little value when used with alphanumeric fields, an understanding of the concept is essential before attempting to use COMP and INDEX data items in tables. The software automatically synchronizes all COMP and INDEX usage data items, and will most probably alter the size of any table (often drastically) that contains these data types. Consider the following illustration of a synchronized data item being mapped by the software:

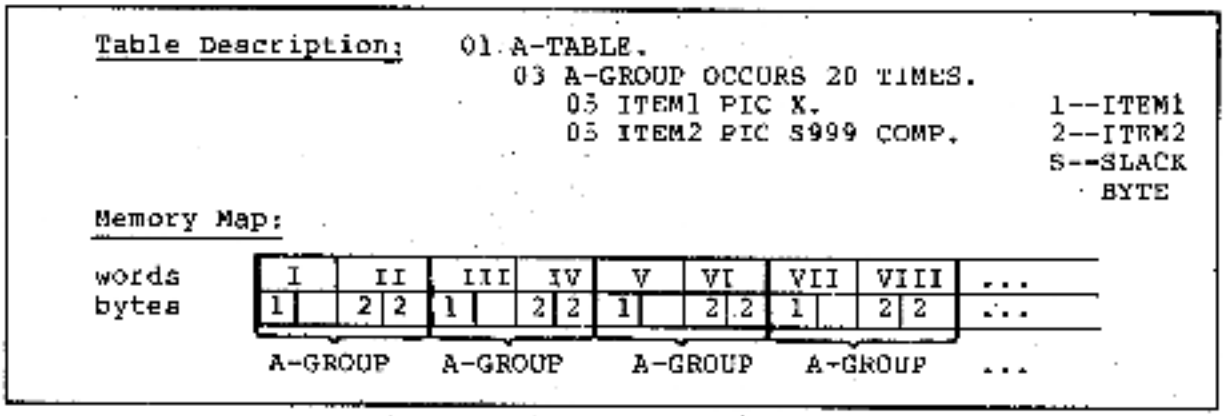


Figure 5-3
Synchronized COMP Item in a Table

Since the software synchronizes the ITEM2 fields (COMP), these fields each occupy a single word in memory; thus, a slack byte follows each occurrence of ITEM1. Each repetition of A-GROUP consumes four bytes of memory -- one byte for ITEM1, one byte for the slack byte, and two bytes for ITEM2. A-TABLE, then, requires 80 bytes of memory (20 elements of four bytes each).

Now, consider the effect of adding a 1-byte field to A-TABLE. If we place the field between ITEM1 and ITEM2, it will take the space formerly occupied by the slack byte. This has the effect of adding a data byte but leaving the size of the table unchanged. Consider the following illustration:

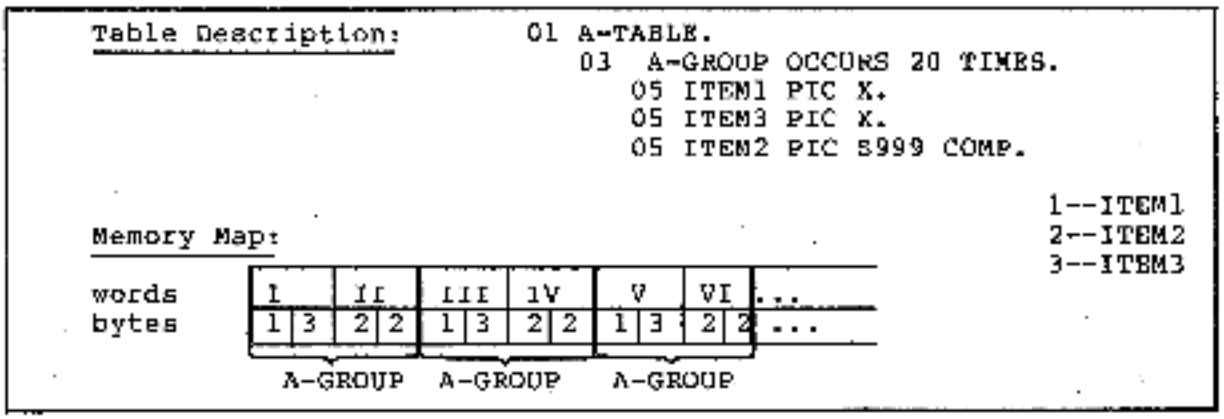


Figure 5-4
Adding a Field without Altering the Table Size

If, however, we place the 1-byte field after ITEM2, it has the effect of adding its own length plus another slack byte. Now, each element requires six full bytes and the complete table consumes 120 bytes of memory (6X20)! This is due to the fact that the first repetition of ITEM1 falls on an even byte and, in order to keep the mapping of each A-GROUP element the same, the software allocates each successive repetition of ITEM1 to an even byte address. Thus, it assigns ITEM3 to the even byte of the third word and adds a slack byte to guarantee that the next element begins on an even byte. Consider the following illustration:

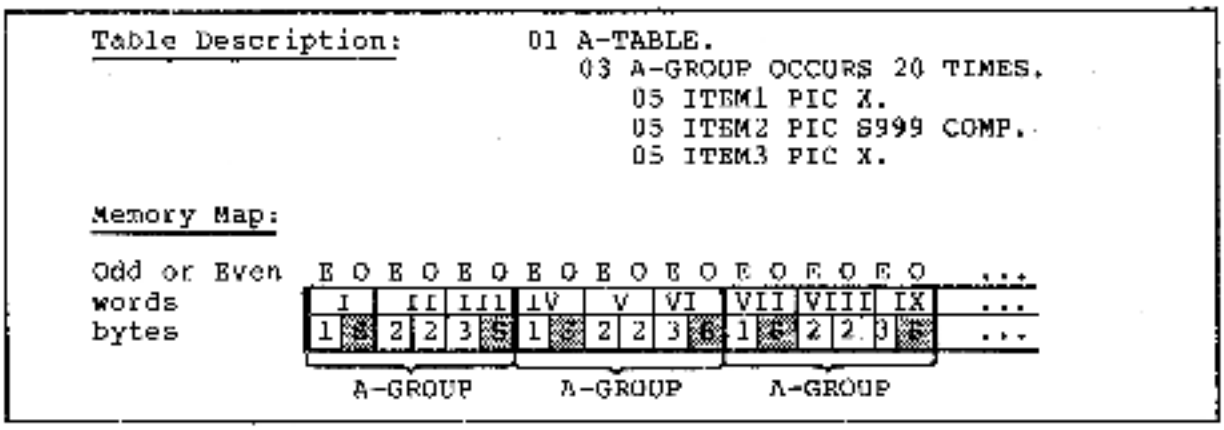


Figure 5-5
Adding One Byte which Adds Two Bytes to the Element Length

NOTE

The illustrations in this section show each byte with an even address (E) as the leftmost byte, and each byte with an odd address (O) as the rightmost byte. (The two bytes, odd and even, are reversed in actual memory.)

If, however, we use a FILLER byte to force the first allocation of ITEM1 to occur on an odd byte, A-GROUP again requires only four bytes and no slack bytes. Figure 5-6 illustrates this. Since the FILLER item occupies the even byte of the first word, ITEM1 falls on an odd byte. The software requires that each repetition of ITEM1 must be an even number of bytes in length in order to guarantee that the synchronized item(s) will map onto word boundaries. No slack bytes are needed and A-GROUP elements are again only four bytes long, and A-TABLE requires only 81 bytes.

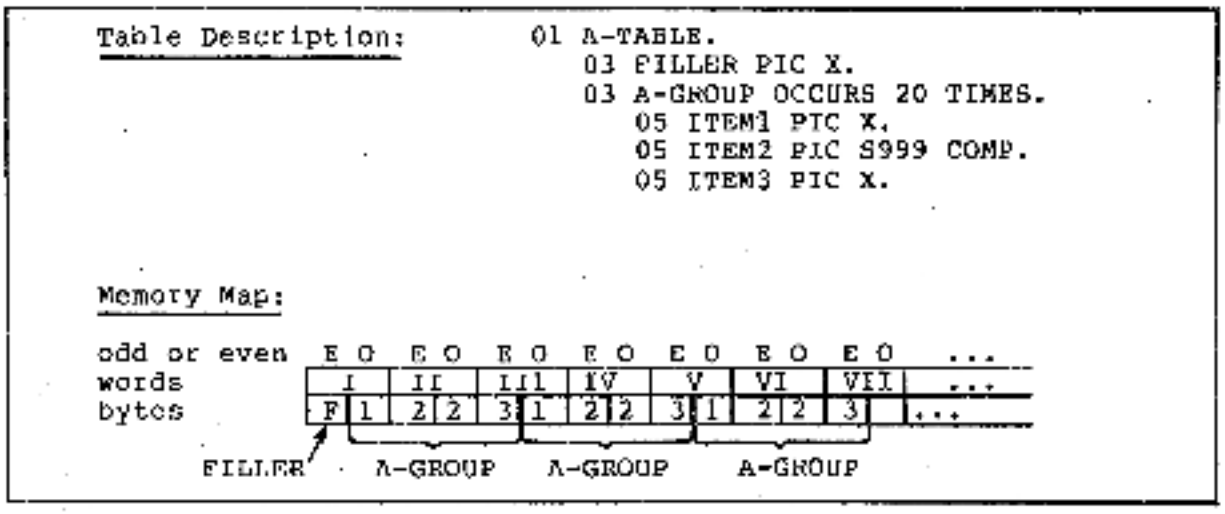


Figure 5-6
Forcing an Odd Address By Adding a 1-Byte FILLER Item to the Head of the Table

If we try to force ITEM1 onto an odd byte with a SYNCHRONIZED RIGHT clause, the software maps ITEM1 into the odd byte, but prohibits all repetitions of the element from using the even byte. Thus, the first repetition of A-GROUP has a slack byte at its beginning and, so that the next element can begin (with a slack byte) at an even address, another slack byte (odd) following ITEM3. This expands the element length to six bytes and the table length to 120 bytes.

Table Description:

```
01 A-TABLE.
   03 A-GROUP OCCURS 20 TIMES.
      05 ITEM1 PIC X SYNCHRONIZED RIGHT.
      05 ITEM2 PIC S999 COMP.
      05 ITEM3 PIC X.
```

Memory Map:

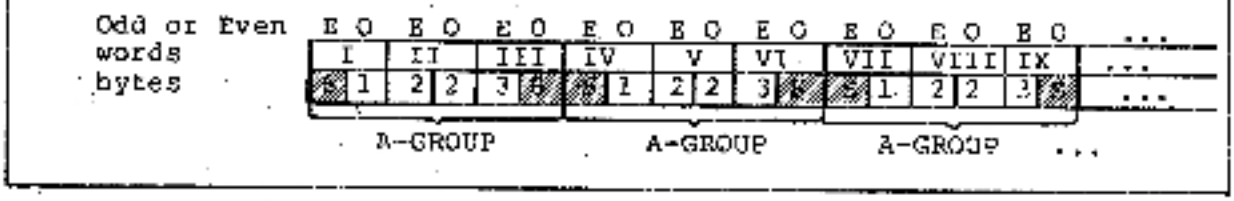


Figure 5-7
The Effect of a SYNCHRONIZED RIGHT Clause Instead of a FILLER Item as shown in Figure 5-6

To determine how the software will map a given table, apply the following two rules:

1. The software maps all items in the first repetition of a table element into memory words as with any item properly defined with a data description, obeying any implicit or explicit synchronization requirements.

2. If the first repetition contains any elementary items with implicit or explicit synchronization, the software maps each successive repetition of the element into memory words in the same way as the first repetition. It does this by adding one slack byte, if necessary, to make the size of the element even.

5.3.1 Initializing Tables

If a table contains only DISPLAY items, it can be set to any desired initial value (initialized). To initialize a table, simply specify a VALUE phrase on the record level preceding the item containing the OCCURS clause. The sample data definitions, below, will set up initialized tables:

Table Description:

```
01 A-TABLE VALUE IS "JANFEBMARAPRMAY
-      JUNJULAUGSEPOCTNOVDEC".
   03 MONTH-GROUP PIC XXX USAGE DISPLAY
                  OCCURS 12 TIMES.
```

Memory Map:

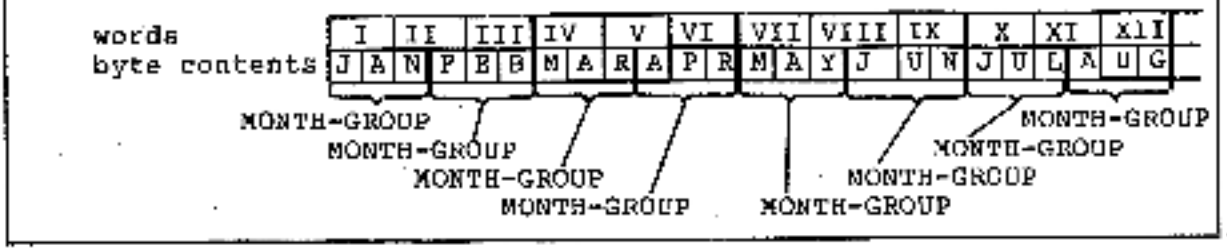


Figure 5-8
Initializing Tables

Often a table is too long to initialize with a single literal, or it contains items that cannot be initialized (numeric, alphanumeric, or COMP). These items can be individually initialized by redefining the group level preceding the level that contains the OCCURS clause. Consider the following sample table descriptions:

Table Description:

```
01 A-RECORD-ALT.
   05 FILLER PIC XX VALUE "AX".
   05 FILLER PIC 99 COMP VALUE 1.
   05 FILLER PIC XX VALUE "BX".
   05 FILLER PIC 99 COMP VALUE 2.
   .
   .
   .
01 A-RECORD REDEFINES A-RECORD-ALT.
   03 A-GROUP OCCURS 26 TIMES.
      05 ITEM1 PIC X.
      05 ITEM2 PIC S99 COMP.
```

Memory Map:

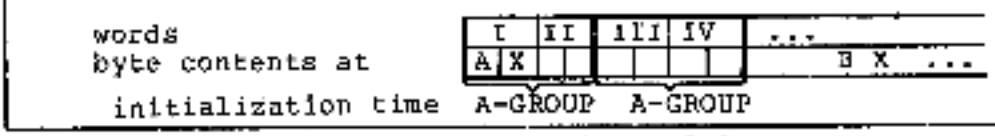


Figure 5-9
Initializing Mixed Usage Fields

In the preceding example, the slack bytes in the alphanumeric fields (ITEM1) are being initialized to X.

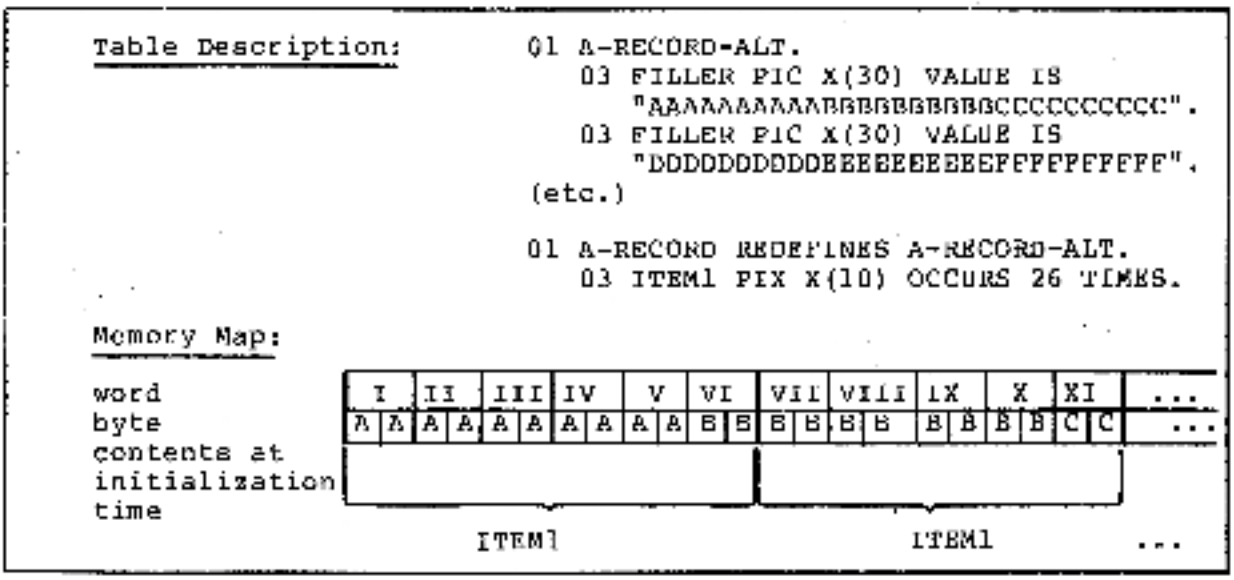


Figure 5-10
Initializing Alphanumeric Fields

In the preceding example, each FILLER item initializes three 10-byte table elements.

When redefining or initializing table elements, allow space for any slack bytes that may be added due to synchronization (implicit or explicit). The slack bytes do not have to be initialized; however, they may be and, if initialized to an uncommon value, they may even serve as a debugging aid for situations such as a statement referring to the record level above the OCCURS clause or another record redefining that level. Sometimes the length and format of table items are such that they would best be initialized by statements in the Procedure Division.

Once the OCCURS clauses have established the necessary tables, the program must be able to access the elements of those tables individually. Subscripting and indexing are the two methods provided by COBOL for accessing individual elements.

5.4 SUBSCRIPTING AND INDEXING

To refer to a particular element within a table, simply follow the name of the desired element with a parenthesized subscript or index. A subscript is an integer or a data-name that has an integer value; the integer value represents the desired occurrence of the element -- an integer value of 3, for example, refers to the third occurrence of the element. An index is a data-name that has been named in an INDEXED BY phrase in the OCCURS clause.

5.4.1 Subscripting with Literals

A literal subscript is simply a parenthesized integer whose value represents the occurrence number of the desired element. In figure 5-11, the literal subscript in the MOVE instruction (2) causes the software to move the contents of the second element of the table, A-TABLE, to I-RECORD.

Table Description	01 A-TABLE. 03 A-GROUP PIC X(5) OCCURS 10 TIMES.
Procedural Instruction	MOVE A-GROUP(2) TO I-RECORD.

Figure 5-11
Literal Subscripting

If the table has more than one level (or dimension), follow the name of the desired item with a list of subscripts, one for each OCCURS clause to which the item is subordinate. The first subscript in the list applies to the first OCCURS clause to which the item is subordinate. (This is the most encompassing level -- A-GROUP in the following example.) The second subscript in the list applies to the next most encompassing level, and the last subscript applies to the lowest level OCCURS clause being accessed (or the desired occurrence number of the item named in the procedural instruction -- ITEM5 in the following example).

Consider Figure 5-12; the subscripts (2,11,3) in the MOVE instruction cause the software to move the third repetition of ITEM5 in the eleventh repetition of ITEM3 in the second repetition of A-GROUP to I-FIELD5. (For illustration simplicity, I-FIELD5 is not defined.) (ITEM5(1,1,1) would refer to the first occurrence of ITEM5 in the table and ITEM5(5,20,4) would refer to the last occurrence of ITEM5.)

Table Description	01 A-TABLE. 03 A-GROUP OCCURS 5 TIMES. 05 ITEM1 PIC X. 05 ITEM2 PIC 99 COMP OCCURS 20 TIMES. 05 ITEM3 OCCURS 20 TIMES. 07 ITEM4 PIC X. 07 ITEM5 PIC XX OCCURS 4 TIMES.
Procedural Instruction	MOVE ITEM5(2, 11, 3) TO I-FIELD5.

Figure 5-12
Subscripting a Multi-Dimensional Table

NOTE

Since ITEM5 is not subordinate to ITEM2, an occurrence number for ITEM2 is not permitted in the subscript list.

Figure 5-13 summarizes the subscripting rules for each of the above items and shows the size of each field in bytes.

NAME OF FIELD	NUMBER OF SUBSCRIPTS REQUIRED TO REFER TO THE NAMED FIELD	SIZE OF FIELD
A-TABLE	NONE	1110
A-GROUP	ONE	222
ITEM1	ONE	1*
ITEM2	TWO	2
ITEM3	TWO	9
ITEM4	TWO	1
ITEM5	THREE	2
* Plus a slack byte		

Figure 5-13
Subscripting Rules for a
Multi-Dimensional Table

5.4.2 Operations Performed by the Software

When a literal subscript is used to refer to an item in a table, the software performs the following steps to determine the exact address of the item:

1. The compiler converts the literal to a 1-word binary value.

2. The compiler range checks the subscript value (the value must not be less than 1 nor greater than the number of repetitions specified by the OCCURS clause) and prints a diagnostic message if the value is out of range.

3. The compiler decrements the value of the subscript by 1 and multiplies it by the size of the item that contains the OCCURS clause corresponding to this subscript, thus forming an index value; it then stores this value, plus the literal subscript, in the object program.

4. At run time, for a fixed length table, the RTS adds the index value (from 3 above) to a base address, thus determining the address of the desired item. For a variable length table reference, the procedure for fixed length tables is preceded by the procedure described in Section 5.4.6.

5.4.3 Subscripting with Data-Names

As discussed earlier in this section, subscripts may also be specified using data-names instead of literals. To use a data-name as a subscript, simply define it as a numeric integer (COMP or DISPLAY). It may be signed, but the sign must be positive at the time it is used as a subscript.

The sample subscripts in figure 5-14 refer to the same element accessed in Figure 5-12, (2, 11, 3).

Data Descriptions of Subscript data-names	01 KEY1 PIC 99 USAGE DISPLAY. 01 KEY2 PIC 99 USAGE COMP. 01 KEY3 PIC S99.
Procedural Instructions	MOVE 2 TO KEY1. MOVE 11 TO KEY2. MOVE 3 TO KEY3. GO TO TABLERTN. TABLERTN. MOVE ITEM5(KEY1 KEY2 KEY3) TO 1-FIELD5.

Figure 5-14
Subscripting with Data-Names

5.4.4 Operations Performed by the RTS

When a data-name subscript is used to refer to an item in a table, the RTS performs the following steps at run time:

1. If the data-name's data type is DISPLAY, the software converts it to a 1-word binary value.

2. For fixed length tables, the software range checks the subscript value (the value must not be less than 1 nor greater than the number of repetitions specified by the OCCURS clause) and terminates the image (with a diagnostic message) if it is out of range. For variable length tables, the procedure described in Section 5.4.6 is followed.

3. The software decrements the value of the subscript by 1 and multiplies it by the size of the item that contains the OCCURS clause corresponding to this subscript, thus forming an index value.

4. The software adds the index value (from 3 above) to a base address, thus determining the address of the desired item.

5.4.5 Subscripting with Indexes

The same rules apply for the specification of indexes as apply to subscripts except that the index must be named in the INDEXED BY phrase of the OCCURS clause.

An index-name item (an item named in the INDEXED BY phrase of the OCCURS clause) has the ability to hold an index value. (The index value is the product formed in step 3 of the operations performed by the software for literal or data-name subscripts -- the relative location, within the table, of the desired item.)

The compiler allocates a 2-part data item for each name that follows an INDEXED BY phrase. These index-name items cannot be accessed as normal data items; they cannot be moved about, redefined, written to a file, etc. However, the SET verb can change their values and relation tests can examine their values. One part of the 2-part index-name item contains a subscript value and the other part contains an index value. Consider the following illustration:

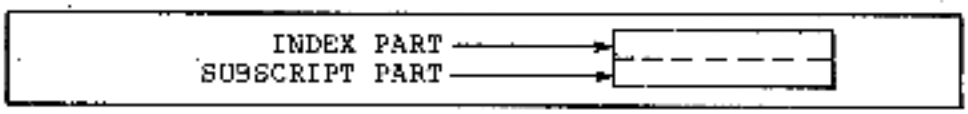


Figure 5-15
Index-Name Item

Whenever a SET statement places a new value in the subscript part, the software performs an index value computation and stores the result in the index part. Only the subscript part of the item acts as a sending or receiving field. The index part is never altered by any other operation and is never moved to another item. It is used only when the index-name is used as an index referring to a table item. The sample MOVE statement in Figure 5-16 would move the contents of the third repetition of A-GROUP to I-FIELD. (For illustration simplicity, once again, I-FIELD is not defined.)

Table Description	01 A-TABLE. 03 A-GROUP OCCURS 5 TIMES INDEXED BY IND-NAME.
Procedural Instructions	SET IND-NAME TO 3. MOVE A-GROUP (IND-NAME) TO I-FIELD.

Figure 5-16
Subscripting With Index-name Items

5.4.6 Operations Performed by the RTS

The RTS performs the following steps when it executes the SET statement:

1. The RTS converts the contents of the sending field of the SET statement to a 1-word binary value.

2. The RTS range checks the value (the value must not be less than 1 nor greater than the number of repetitions specified in the OCCURS clause) and terminates the image with a diagnostic message if it is out of range.

3. The RTS decrements the value by 1 and multiplies it by the size of the item that contains the OCCURS clause, thus forming an index value.

For fixed length tables, once the SET statement has been executed and the software has encountered the index-name item as an index, it only has to add the index value (from 3 above) to a base address to determine the address of the desired item. Since this is the only action performed, the execution speed of a procedural statement with an indexed data-name is equivalent to a reference with a literal subscript.

For a variable length table, when the index-name is encountered as an index, the procedure described in Section 5.4.6 is invoked before following the fixed length table logic. However, the SET statement itself is not impacted by the fixed/variable characteristic of the associated table.

VAX-11 COBOL-74 initializes the value of all index-name items to a subscript value of 1 (index value of 0), hence an attempt to use an index-name item as an index before it has been the receiving field of a SET verb will not result in an out-of-range termination.

NOTE

Initialization of index-name items is an extension to the ANSI COBOL standards. Users concerned with writing COBOL programs that adhere to standard COBOL should not rely on this feature.

5.4.7 Relative Indexing

To perform relative indexing, when referring to a table item, simply follow the index-name with a plus or minus sign and an integer literal. Relative indexing, albeit easy to use, causes additional overhead to be generated each time a table item is referenced in this fashion. At compile time, the compiler has to compute the index value corresponding to the specified literal; and transfer this index value to the object file. At run time, the index value for the literal is added to (+) or subtracted from (-) the index value of the index-name.

The resulting index value is stored in a temporary location. The RTS adds this temporary index value to the base address of the table to determine the address of the desired table item. At this point, a range check is performed on the temporary index value to insure that the resulting index is within the permissible range for the table.

For fixed length tables, this index manipulation is relatively straightforward. The size of the table is known at compilation time, and this size is passed along to the RTS in the object file. A simple compare against this fixed value is all that is required to determine if a given index value is within the permissable range for the table.

For a variable length table, however, the process is more involved. The current number of occurrences (data-name-1) for the table must be determined and range checked; the index value corresponding to the current number of occurrences must be calculated; then the temporary index value must be range checked using the current number of occurrence's index value.

The run-time overhead required for the relative indexing of variable length tables is significantly greater than that required for fixed length tables. In either case, the index portion of the index-name is not altered. If any of the range checks reveals an illegal (out of range) value, execution is terminated with an apropriate error message.

The sample MOVE instruction in Figure 5-17 moves the fourth repetition of A-GROUP to I-FIELD if IND-NAME has not been altered with a SET verb.

```
MOVE A-GROUP(IND-NAME + 3) TO I-FIELD.
```

Figure 5-17
Relative Indexing

The actual operation of accessing a table element is shorter at run time since the compiler has calculated the index value of the literal at compile time and has stored it in the object program ready for use. Relative indexing, therefore, involves two additions and a range check dat run time. It leaves the index-name item unaltered.

5.4.8 Index Data Items

Often a program will require that the value of an index-name item be stored outside of that item. It is for this purpose that VAX-11 COBOL-74 provides the index data item.

Index data items are 1-word binary integers with implicit synchronization. (The 1-word size corresponds to the subscript part of the index-name item.) They must be declared with a USAGE IS INDEX phrase and they may be modified (explicitly) only by the SET statement.

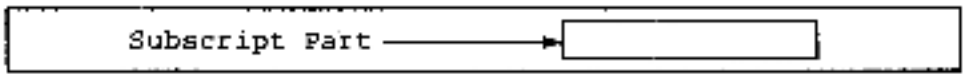


Figure 5-18
Index Data Item

Since index data items are considered to contain only the subscript part of an index-name item, when a SET statement "moves" an index-name item to an index data item, only the subscript part is moved. Likewise, when a SET statement "moves" an index data item to an index-name item, a new index value is computed by the software. This is done to guarantee that an index-name item will always contain a good index value.

The only advantage gained by using index data items over numeric, COMP items is that the data description is shorter, easier to write, and more self-documenting. Further, the restrictions placed on access to index items may be useful in debugging the program.

5.4.9. The SET Statement

The SET statement alters the value of index-name items and copies their value into other items. When used without the UP BY/DOWN BY clause, it functions like a MOVE statement. Figure 5-19 illustrates the legal data movements that the SET statement can perform.

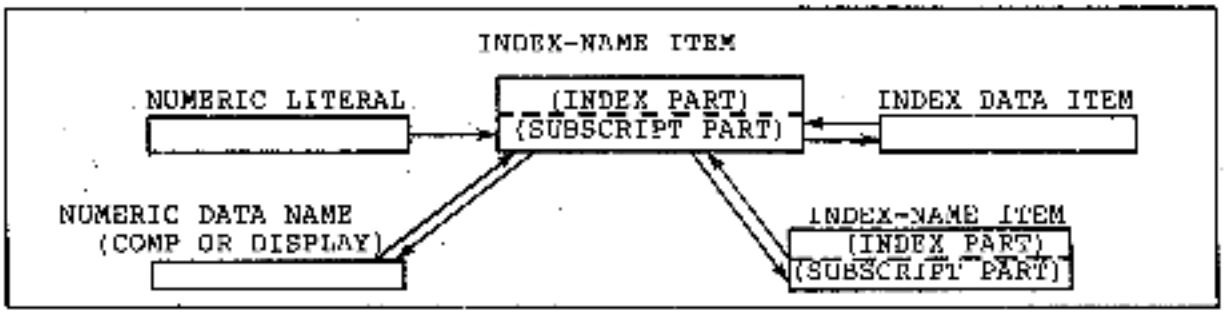


Figure 5-19
Legal Data Movement with the SET Statement

The SET statement may be used with the UP BY/DOWN BY clause to alter the value of an index-name item arithmetically. The numeric literal is added to (UP BY) or subtracted from (DOWN BY) the subscript part, and the index part is recalculated by the software after the appropriate range check against the number of repetitions for the table. The SET statement is not affected by whether the table is fixed or variable length.

5.4.10 Referencing a Variable-Length Table Element at Run Time

At run time, when a procedural reference involves an element in a variable length table, the following procedure is used:

1. Determine the number of occurrences in the table (the value contained in data-name-1), and verify its legality.

   (integer-1 <= data-name-1 <= integer-2)

2. Verify that the subscript is within the legal range.

   (subscript <= data-name-1)

If any of the above checks fails, execution is terminated with an appropriate error message.

5.4.11 Referencing a Dynamic Group at Run Time

A dynamic group is defined as a group item that contains a subordinate item that is a variable length table. At run time, when a dynamic group is referenced, the following procedures are followed:

1. The number of occurrences of the subordinate variable length table is determined, and checked for legality; i.e., integer-1<=data-name-1<=integer-2. If this check fails, execution terminates and the appropriate error message is issued.

2. The size of the dynamic group is calculated. The number of occurrences of the variable length table (data-name-1) is multiplied by the size of one table entry. The resulting number is then added to the fixed size of the dynamic group.

NOTE

The fixed size of a dynamic group is the size of the group up to but not including the variable length table.

5.4.12 The SEARCH Verb

The SEARCH verb has two formats: Format 1, which performs a sequential search of the specified table beginning with the current index setting; and Format 2 which performs a selective (binary) search of the specified table, beginning with the middle of the table.

Both formats allow the programmer to specify imperative statements within the SEARCH verb. At run time, an imperative statement contained within a search verb is executed only when one of the exit paths (success or failure) is taken.

The failure path is defined either explicitly by the AT END statement, in which case the imperative statement which follows it is executed; or by default, in which case control is passed to the next procedural sentence. In either case (success or failure), after an imperative statement is executed, control is passed to the next procedural sentence.

5.4.13 The SEARCH Verb - Format 1

Format 1 directs the RTS to search the indicated table sequentially. The OCCURS clause for the table being searched must contain the INDEXED by phrase. Unless otherwise specified in the SEARCH statement, the first index is the controlling index for the table search. The search begins with the current index setting, and progresses through the table, augmenting the index by one as each occurrence is interrogated. If any of the specified conditions is true (success), the associated imperative statement is executed; the search exits; and the index remains at the current setting.

If the possible number of occurrences for the table is exhausted before any of the specified conditions are met, the specified failure exit path is taken. That is, either the AT END exit path (if specified) is taken, or control is passed to the next procedural sentence.

Figure 5-20 contains an example of using the SEARCH verb to search a table in a serially.

Associated with Format 1 is the optional VARYING phrase. This phrase can be specified by using any of the following methods:

1. default - phrase omitted
2. VARYING index-name-n
3. VARYING identifier-2
4. VARYING index-name-2

NOTE

The following is true regardless of which of the above methods is used.

a. An index name associated with the table is methodically augmented by one, by the RTS, for each cycle of the serial search. This controlling index, when compared to the allowable number of occurrences for the table, dictates the permissible range of search cycles at run time. When an exit occurs (success or failure), this index remains at the current setting.

b. The RTS will not initialize the index when the search begins. It is the programmers responsibility to insure that the initial index setting is the appropriate one. The RTS will begin processing the table with the setting it finds when the search is initiated.

When method 1 is used, the first index name (index-name-1) associated with the table is used as the controlling index. Only this index is set to consecutive values by the RTS serial search processor. See Figure 5-20, Example 2, for an example of using method 1.

When method 2 is used, index-name-n is any index that is associated with the table being searched. It becomes the controlling index for the table. It alone is set to consecutive values by the RTS search processor. See Figure 5-20, Example 3, for an example of using method 2.

When method 3 is used, identifier-2 is augmented by one each time the first index (controlling index) for the table is augmented by one. Identifier-2 is not a substitute index. It merely allows the programmer to maintain an additional pointer to elements within a table. See Figure 5-20, Example 4, for an example of method 3.

When method 4 is used, index-name-2 is an index that is associated with a table other than the one being searched. Each time the controlling index (1st index for the table) of the searched table is augmented, index-name-2 is also augmented. See Figure 5-20, Example 5.

5.4.14 The SEARCH Verb - Format 2

Format 2 is used to direct the RTS to search the indicated table selectively. The selective (binary) search is predicated upon the ASCENDING/DESCENDING KEY attributes of the table being searched. Therefore, an ASCENDING and/or DESCENDING KEY(s) must be specified in the OCCURS clause that defines the table, to inform the RTS that the keys are stored within the table in ascending or descending order.

The INDEXED BY phrase must also be specified. When the binary search is executed, the RTS uses the first or only index associated with the table as the controlling index for the search. The selective (binary) search is implemented in the RTS as follows:

1. The RTS examines the range of permissible values for the index of the table being searched; selects the median value; and assigns this median value to the index.

2. The RTS then proceeds to process the sequence of simple tests for equality, beginning with the first, with the index set to the median value.

3. If all of the tests for equality are true (success), the search is terminated; the associated imperative statement is executed; the search exits; and the index retains its current value.

4. If any of the tests for equality is false, the following results occur.

   a. The RTS determines if all of the possible occurrences for the table have been tested. If the table has been exhausted, the imperative statement which accompanies the AT END statement (if specified) is executed. In either case, control is passed to the next procedural statement.

   b. The RTS will now determine which half of the table is to be eliminated from further consideration. This determination is predicated on whether the key being tested is in ascending or descending order, and whether the test failed because of a greater than or less than comparison. For example, if the key values being tested are stored in ascending order, and the median table element being tested is greater than the value being tested for equality, the RTS will assume that all key elements following the one tested are also greater than the value being tested for equality. Therefore, the lower half of the table, those items which follow the current index setting, are no longer in contention.

   c. Once the direction of search is determined, half of the table is eliminated from further consideration. A new range of permissible index values is computed from the remaining half of the table.

   d. Processing begins all over again from step 1.

See Figure 5-20, Example 6, for an example of searching a table using Format 2 of the SEARCH verb.

```
FED-TAX-TABLES.
02  ALLOWANCE-DATA.
    03  FILLER                      PIC X(70) VALUE
        "0001440
        "0202880
        "0304320
        "0405760
        "0507200
        "0608640
        "0710080
        "0811520
        "0912960
        "1014400".
02  ALLOWANCE-TABLE REDEFINES ALLOWANCE-DATA.
    03  FED-ALLOWANCES OCCURS 10 TIMES
        ASCENDING KEY IS ALLOWANCE-NUMBER
        INDEXED BY IND-1.
        04  ALLOWANCE-NUMBER        PIC XX.
        04  ALLOWANCE               PIC 999V99.
02  SINGLES-DEDUCTION-DATA.
    03  FILLER                      PIC X(112) VALUE
        "0258006700800016
        "0670011900067220
        "1150018300163223
        "1838024800319621
        "2400027900439326
        "2790034600540730
        "3060049999741736".
02  SINGLES-DEDUCTION-TABLE REDEFINES SINGLES-DEDUCTION-DATA.
    03  SINGLES-TABLE OCCURS 7 TIMES
        ASCENDING KEY IS S-MIN-RANGE S-MAX-RANGE
        INDEXED BY IND-2, TEMP-INDEX.
        04  S-MIN-RANGE             PIC 999V99.
        04  S-MAX-RANGE             PIC 999V99.
        04  S-TAX                   PIC 99V99.
        04  S-PERCENT               PIC V99.
02  MARRIED-DEDUCTION-DATA.
    03  FILLER                      PIC X(119) VALUE
        "0490009600480008017
        "0960017300008162P
        "1730026400023561?
        "2640034600390325
        "3460043300595328
        "4330053000083A932
        "5P00999999910533336".
02  MARRIED-DEDUCTION-TABLE REDEFINES MARRIED-DEDUCTION-DATA.
    03  MARRIED-TABLE OCCURS 7 TIMES
        ASCENDING KEY IS M-MIN-RANGE M-MAX-RANGE
        INDEXED BY IND-0, IND-3.
        04  M-MIN-RANGE             PIC 999V99.
        04  M-MAX-RANGE             PIC 999V99.
        04  M-TAX                   PIC 999V99.
        04  M-PERCENT               PIC V99.
TEMP-INDEX                           USAGE INDEX.
```

Figure 5-20
Example of Using SEARCH
To Search a Table

Example 1

```
SINGLE.
        IF TAXABLE-INCOME < 02499
                GO TO END-FED-COMP.
        SET IND-2 TO 1.
        SEARCH SINGLES-TABLE VARYING IND-2 AT END
                GO TO TABLE-2-ERROR
          WHEN TAXABLE-INCOME = S-MIN-RANGE(IND-2)
                MOVE S-TAX(IND-2) TO FED-TAX-DEDUCTION OF
                        OUTPUT-MASTER
                GO TO STORE-FED-TAX
          WHEN TAXABLE-INCOME < S-MAX-RANGE(IND-2)
                SUBTRACT S-MIN-RANGE(IND-2) FROM TAXABLE-INCOME
                MULTIPLY TAXABLE-INCOME BY S-PERCENT(IND-2) ROUNDED
                ADD TAXABLE-INCOME TO FED-TAX-DEDUCTION OF
                        OUTPUT-MASTER.
```

Example 2

```
SINGLE.
        IF TAXABLE-INCOME < 02499
                GO TO END-FED-COMP.
        SET IND-2 TO 1.
        SEARCH SINGLES-TABLE VARYING IND-2 AT END
                GO TO TABLE-2-ERROR
          WHEN TAXABLE-INCOME = S-MIN-RANGE(IND-2)
                MOVE S-TAX(IND-2) TO FED-TAX-DEDUCTION OF
                        OUTPUT-MASTER
                GO TO STORE-FED-TAX
          WHEN TAXABLE-INCOME < S-MAX-RANGE(IND-2)
                SUBTRACT S-MIN-RANGE(IND-2) FROM TAXABLE-INCOME
                MULTIPLY TAXABLE-INCOME BY S-PERCENT(IND-2) ROUNDED
                ADD TAXABLE-INCOME TO FED-TAX-DEDUCTION OF
                        OUTPUT-MASTER.
```

Example 3

```
MARRIED.
        IF TAXABLE-INCOME < 04799
                MOVE ZEROS TO FED-TAX-DEDUCTION OF OUTPUT-MASTER,
                GO TO END-FED-COMP.
        SET IND-3 TO 1.
        SEARCH MARRIED-TABLE VARYING IND-3
                AT END GO TO TABLE-3-ERROR
          WHEN TAXABLE-INCOME = M-MIN-RANGE(IND-3)
        MOVE M-TAX(IND-3) TO FED-TAX-DEDUCTION OF OUTPUT-MASTER,
                GO TO STORE-FED-TAX,
          WHEN TAXABLE-INCOME < M-MAX-RANGE(IND-3)
        MOVE M-TAX(IND-3) TO FED-TAX-DEDUCTION OF OUTPUT-MASTER,
        SUBTRACT M-MIN-RANGE(IND-3) FROM TAXABLE-INCOME ROUNDED,
        MULTIPLY TAXABLE-INCOME BY M-PERCENT(IND-3) ROUNDED,
        ADD TAXABLE-INCOME TO FED-TAX-DEDUCTION
                        OF OUTPUT-MASTER ROUNDED,
                GO TO STORE-FED-TAX.
```

Figure 5-20 (Cont.)
Example of Using SEARCH
To Search a Table

Example 4

```
SINGLE.
        IF TAXABLE-INCOME < 02499
                GO TO END-FED-COMP.
        SET IND-2 TO 1.
        SEARCH SINGLES-TABLE VARYING TEMP-INDEX AT END
                GO TO TABLE-2-ERROR
          WHEN TAXABLE-INCOME = S-MIN-RANGE(IND-2)
                MOVE S-TAX(IND-2) TO FED-TAX-DEDUCTION OF
                        OUTPUT-MASTER
                GO TO STORE-FED-TAX
          WHEN TAXABLE-INCOME < S-MAX-RANGE(IND-2)
                SUBTRACT S-MIN-RANGE(IND-2) FROM TAXABLE-INCOME
                MULTIPLY TAXABLE-INCOME BY S-PERCENT(IND-2) ROUNDED
                ADD TAXABLE-INCOME TO FED-TAX-DEDUCTION OF
                        OUTPUT-MASTER.
```

Example 5

```
SINGLE.
        IF TAXABLE-INCOME < 02499
                GO TO END-FED-COMP.
        SET IND-2 TO 1.
        SEARCH SINGLES-TABLE VARYING IND-0 AT END
                GO TO TABLE-2-ERROR
          WHEN TAXABLE-INCOME = S-MIN-RANGE(IND-2)
                MOVE S-TAX(IND-2) TO FED-TAX-DEDUCTION OF
                        OUTPUT-MASTER
                GO TO STORE-FED-TAX
          WHEN TAXABLE-INCOME < S-MAX-RANGE(IND-2)
                SUBTRACT S-MIN-RANGE(IND-2) FROM TAXABLE-INCOME
                MULTIPLY TAXABLE-INCOME BY S-PERCENT(IND-2) ROUNDED
                ADD TAXABLE-INCOME TO FED-TAX-DEDUCTION OF
                        OUTPUT-MASTER.
```

Example 6

```
FED-DEDUCT-COMPUTATION.
        SET IND-1 TO 1.
        SEARCH ALL FED-ALLOWANCES AT END GO TO TABLE-1-ERROR
          WHEN ALLOWANCE-NUMBER(IND-1) = NR-DEPENDENTS OF
                OUTPUT-MASTER,
          SUBTRACT ALLOWANCE(IND-1) FROM GROSS-WAGE OF OUTPUT-MASTER
                GIVING TAXABLE-INCOME ROUNDED,
        IF MARRITAL-STATUS OF OUTPUT-MASTER = "M"
                GO TO MARRIED.
```

Figure 5-20 (Cont.)
Example of Using SEARCH
To Search a Table

CHAPTER 6

INPUT-OUTPUT PROCESSING

This chapter relates COBOL-74 I/O concepts to the features and requirements of VAX/VMS and Record Management Services (RMS), which is the file and record access subsystem of the VAX/VMS operating system. Rather than being a complete description of RMS and COBOL-74 I/O, this chapter provides a bridge between the concepts and facilities of both. For more detailed information:

1. The **VAX-11 COBOL-74 Language Reference Manual** describes the I/O concepts and statements that are part of the COBOL language.

2. The basic concepts of RMS, such as file organization, access modes, and the physical attributes of file devices, are discussed in the **Introduction to VAX-11 Record Management Services.**

Most of the following discussion relates to file handling. However, input-output is largely device-independent in the VAX-11 system; the logical name translation facility of VAX/VMS allows you to specify I/O in a generalized and flexible way. Therefore, sections of this chapter describe file naming techniques and conventions, relating them to both file-associated I/O statements (like OPEN, READ, and WRITE) and the low-volume I/O statements (ACCEPT and DISPLAY), which in many systems apply only to terminal devices.

FILE ATTRIBUTES

A file is an organized collection of records that is stored and maintained on an accessible medium, such as disk or magnetic tape. Both RMS and COBOL-74 support a variety of devices, file organizations, access methods, and storage techniques. Programs communicate with RMS by using a set of conventions and structures that allow compatibility of file definitions and operations among all users of RMS.

File attributes are characteristics that determine how a file's records are organized and how they can be stored and accessed. The attributes are specified when a file is created, either by a program you write or by the RMS DEFINE Utility.

RMS stores the file attributes in the file itself, so they are available whenever the file is accessed. When a program accesses a file through RMS, it must use the attributes that were defined when the file was created. For example, a program cannot read a sequential file as an indexed file, since the indexes do not exist; it also cannot correctly access a file if it specifies a different blocking factor or record format (fixed-length instead of variable-length, for instance) than was originally defined.

COBOL programs specify file attributes through a combination of statements in the Environment and Data Divisions:

1. The SELECT statement specifies the file organization.
2. File-description-entries specify record blocking and record format.
3. Record-description-entries specify record sizes and can specify record format.

6.1 RECORD FORMAT

The RMS record format determines whether the size of a file's records can vary; and if RMS stores control fields with records on the storage medium.

The compiler determines the record format attribute from a combination of record-description-entries, the RECORD CONTAINS clause, and clauses that specify print-controlled files.

6.1.1 Fixed-length

Files with the fixed-length record format contain records that are all the same size. The compiler generates the fixed-length record format file attribute when:

The file has only one record description, or it has multiple record descriptions, all of which are the same size;

AND

The file description does not contain a "RECORD CONTAINS integer-1 TO integer-2" phrase;

AND

The program does not specify a print-controlled file by using any of the following to refer to the file:

- The ADVANCING phrase in a WRITE statement
- An APPLY PRINT-CONTROL clause in the Environment Division
- A LINAGE clause in the file description

6.1.2 Variable-length

Files with the variable-length record format can contain records that vary in size. RMS stores a record length field before the beginning of each data record.

- For disk files, the record length field is a 2-byte (1-word) binary value that specifies the length of the record in bytes (excluding the record length field).
- For tape files, the record length field is a 4-byte decimal value that specifies the length of the record in bytes (including the record length field).

The compiler generates the variable-length attribute for a file when:

The file has more than one record description, and not all records described for the file are the same size;

OR

the file description contains a "RECORD CONTAINS integer-1 TO integer-2" phrase.

6.1.3 Variable With Fixed-Length Control

Variable with fixed-length control records are similar to variable-length records; however, they contain a fixed-length control field between the record length field and the variable-length data record.

The compiler generates this attribute only for print-controlled files and for files that are opened by a DISPLAY statement; the RTS stores print-control values in the fixed-length control field, which is two bytes long. Section 6.1.1 describes the COBOL language specification of print-controlled files.

6.2 RECORD SIZE

The space needed for a record on a storage medium depends on the record format, the file organization, and the size of the COBOL record description that is used to write the record.

The maximum size of a record depends on record format:

- For fixed-length records, the maximum size is the record size.
- For variable-length records, the maximum size is the size of the file's largest record description in the COBOL program that created the file, plus the number of overhead bytes needed for the storage medium.

In relative files, all records are stored in fixed-length "cells". Cell size is one byte larger than fixed-length record size; the extra byte is a delete flag, which RMS uses to determine if a cell contains a record. For variable-length records, cell size is three bytes larger than the maximum record size; it includes the delete flag and the 2-byte record length field.

In sequential and indexed files, however, variable-length records can save space. RMS stores records contiguously whenever possible; therefore, a variable-length record requires less space than a fixed-length record of maximum size if its length differs from the maximum record size by more than its variable-length overhead.

The size of a data record written by a COBOL program is determined only by the effective size of the record description named in the WRITE statement. In the following example, the first WRITE statement causes 56 bytes to be written to the file specified by ACCOUNT-FILE; the second WRITE statement transfers 42 bytes:

```
FD  ACCOUNT-FILE
.
.
.
01  ACCT-FLAG-REC.
    03  FILLER          PIC X(40).
    04  ACCT-FLAG       PIC 9(2).
01  ACCT-REC.
    03  ACCT-NUM        PIC 9(10).
    03  ACCT-NAME       PIC X(30).
    03  ACCT-LIMIT      PIC 9(6).
    03  ACCT-DISCOUNT   PIC 99V99.
    03  ACCT-DATE       PIC 9(6).
.
.
.
    WRITE ACCT-REC.
.
.
.
    WRITE ACCT-FLAG-REC.
```

If the records are written to a relative file whose maximum record size is 56 bytes, the RTS still transfers 56 bytes and 42 bytes; however, the space needed for each record is 58 bytes, excluding the delete flag. For the 42-byte record, RMS transfers the 2-byte record length field and the 42-byte data record, leaving 14 unused (and inaccessible) bytes, excluding the delete flag.

6.3 RECORD BLOCKING

Record blocking can increase the execution speed of programs that perform many file I/O operations. In general terms, a block is the unit of data transfer between a program and a file storage device. A block can contain one or more records; if it contains many records, I/O speed can increase -- the program requires only one transfer from the device to memory in order to consecutively read records in the same block.

The COBOL language expresses block size in terms of records or characters. The actual size of the unit of data transfer is affected by the storage medium, record format and file organization. Because the term "block" can have several meanings, the rest of this chapter uses terminology that is more specific:

Physical Block

A group of consecutive bytes of data treated as a unit by the storage medium. On magnetic tape, a physical block can vary in size; it is the number of bytes between two interrecord gaps. On disk, a physical block is a 512-byte unit.

A physical block can contain one or more records, or it can contain part of a record; records can span physical block boundaries.

Physical block is synonymous with the VAX-11 RMS term, block.

Bucket

For relative and indexed files, the RMS unit of transfer between storage devices and I/O buffers in memory. A bucket can contain one or more records; however, records cannot span buckets.

Record Unit Size

The storage medium space (in bytes) needed to store a record in a file.

For fixed-length records, record unit size is the record length.

For variable-length records, record unit size is the maximum record length plus the size of the count field.

For variable with fixed-length control records, record unit size is the sum of the maximum record length, the count field size, and the size of the print-control field (two bytes).

In COBOL, you can specify blocking in terms of records or characters, or you can use the compiler's default. The following sections discuss the three methods separately for each file organization. The examples refer to the following samples of COBOL file and record descriptions:

Sample A

```
FD  TEST-FILE
    LABEL RECORDS ARE STANDARD.
01  REC-1          PIC X(100).
01  REC-2          PIC X(511).
```

Sample B

```
FD  TEST-FILE
    BLOCK CONTAINS 50 RECORDS
    LABEL RECORDS ARE STANDARD.
01  REC-1          PIC X(20).
```

Sample C

```
FD  TEST-FILE
    BLOCK CONTAINS 512 CHARACTERS
    LABEL RECORDS ARE STANDARD.
01  REC-1          PIC X(494).
```

6.3.1 Sequential Files on Magnetic Tape

Default

The physical block size is determined when the volume is mounted; if the /BLOCK=n qualifier is not used with the VAX/VMS MOUNT command, the RMS default is 2048 characters.

BLOCK CONTAINS n RECORDS

The compiler computes the physical block size as n multiplied by the record unit size.

Example:

Sample	Physical block size
B	1000 bytes (50*20)

BLOCK CONTAINS n CHARACTERS

If n is less than the record unit size, the compiler ignores n and uses the record unit size as the physical block size. Otherwise, the physical block size equals n. Records cannot span physical blocks; therefore, a physical block can contain only complete records (regardless of record format). A physical block is transferred (written) to the magnetic tape device when the program tries to add a record that cannot fit into the I/O buffer; the unwritten record begins the next physical block.

Example:

Sample	Physical block size
C	512 bytes (18 unused)

6.3.2 Sequential Files on Disk

Records are packed together in each physical block; there are no unused bytes in any block, and the records can span block boundaries.

Default

The RMS default determines record blocking for sequential disk files.

BLOCK CONTAINS n RECORDS

The compiler computes the unit of data transfer, in terms of 512-byte physical blocks, as follows:

Unit of data transfer =
(n*record unit size/512), rounded up

Example:

Sample	Unit of data transfer
B	2 = (50*20/512), rounded up

BLOCK CONTAINS n CHARACTERS

The compiler computes the unit of data transfer, in terms of 512-byte physical blocks, as follows:

Unit of data transfer = n/512, rounded up

Example:

Sample	Unit of data transfer
C	1 = 512/512, rounded up

6.3.3 Relative Files

In each of the following methods for computing bucket size, one byte is added to the record unit size. RMS adds one byte to each cell in a relative file to indicate whether the cell contains a record or is empty. Bucket size is expressed in terms of 512-byte physical blocks.

The bucket size is a file attribute; therefore, each time you access the file, you must specify it the same way as when the file was created.

The following examples refer to the COBOL samples presented in Section 6.3.

Default

The compiler tries to make the bucket size as small as possible by computing it as follows:

Bucket size = ((1+record unit size)/512), rounded up

Example:

Sample	Bucket size
A	2 = ((1+(2+511))/512), rounded up

BLOCK CONTAINS n RECORDS

The compiler computes the bucket size as follows:

Bucket size = (n*(1+record unit size)/512), rounded up

Example:

Sample	Bucket size
B	3 = (50*(1+20)/512), rounded up

BLOCK CONTAINS n CHARACTERS

The compiler computes the bucket size as follows:

Bucket size = n/512

Where:

- n must equal or exceed (1+record unit size). If it is less than that quantity, the compiler issues a warning diagnostic and uses the default method to compute the bucket size.
- n must be a multiple of 512. If not, the compiler issues a warning diagnostic and rounds n up to the next multiple of 512.

Example:

Sample	Bucket size
C	1

6.3.4 Indexed Files

Each of the methods for computing bucket size for indexed files considers overhead bytes for each record and bucket:

Record overhead = 7 bytes
Bucket overhead = 15 bytes

The bucket size is a file attribute; therefore, each time you access the file, you must specify it the same way as when the file was created.

In each of the following methods, bucket size is expressed in terms of 512-byte physical blocks. Again, the examples refer to the COBOL samples presented in Section 6.3.

Default

The compiler tries to make the bucket size as small as possible by computing it as follows:

Bucket size = ((15+(7+record unit size))/512), rounded up

Example:

Sample	Bucket size
A	2 = ((15+(7+(2+511)))/512), rounded up

BLOCK CONTAINS n RECORDS

The compiler computes the bucket size as follows:

Bucket size = ((15+(7+record unit size)*n)/512), rounded up

Example:

Sample	Bucket size
B	4 = ((15+(7+20)*50)/512), rounded up

BLOCK CONTAINS n CHARACTERS

The compiler computes the bucket size as follows:

Bucket size = n/512

Where:

- n must equal or exceed (15+(7+record unit size)). If it is less than that quantity, the compiler issues a warning diagnostic and uses the default method to compute the bucket size.
- n must be a multiple of 512. If not, the compiler issues a warning diagnostic and rounds n up to the next multiple of 512.

Example:

Sample	Bucket size
C	2
	n*(512) is less than the minimum required by the first rule; therefore, the compiler uses the default method.

6.4 CURRENT RECORD AREA

A file's current record area is the location in which records are available to a COBOL program; it is defined by the record descriptions that follow the file description.

COBOL I/O statements appear to transfer data directly between a file and its current record area. Actually, I/O statements transfer data between the current record area and the file's I/O buffer. RMS manages the I/O buffers and transfers data between them and files.

6.4.1 Effects on Output Operations

The current record area includes the total area described by all of a file's record descriptions: it is as large as the largest record described for the file.

The size of a record written by a COBOL program, however, is determined by the record description named in the WRITE or REWRITE statement.

6.4.2 Effects of Input Operations

The RTS does not clear the current record area before executing a READ operation; therefore, the contents of the current record area after a READ are determined by the length and contents of the record. For example, when the program reads a record that is smaller than the largest record described for a file, the operation does not change the area beyond the end of the incoming record.

Consider an example in which the current record area contains 20 characters from the first record read from a file. If the next READ returns a 12-character record, the remaining 8 character positions are not changed:

Current record area after first READ:	0239394CABINET, FILE
Contents of next record:	6627402CHAIR
Current record area after next READ:	6627402CHAIRET, FILE

It is not considered a good COBOL programming practice to depend on this condition.

6.4.3 Sharing Record Areas

The compiler normally allocates storage space separately for each file's current record area. This method of current record area allocation has two potentially undesirable effects:

1. Each file's current record area requires storage in the program's Data Division. If the number of files and the sizes of their records are large, Data Division size could approach COBOL-74's Data Division size limitation, which is 65,535 bytes.

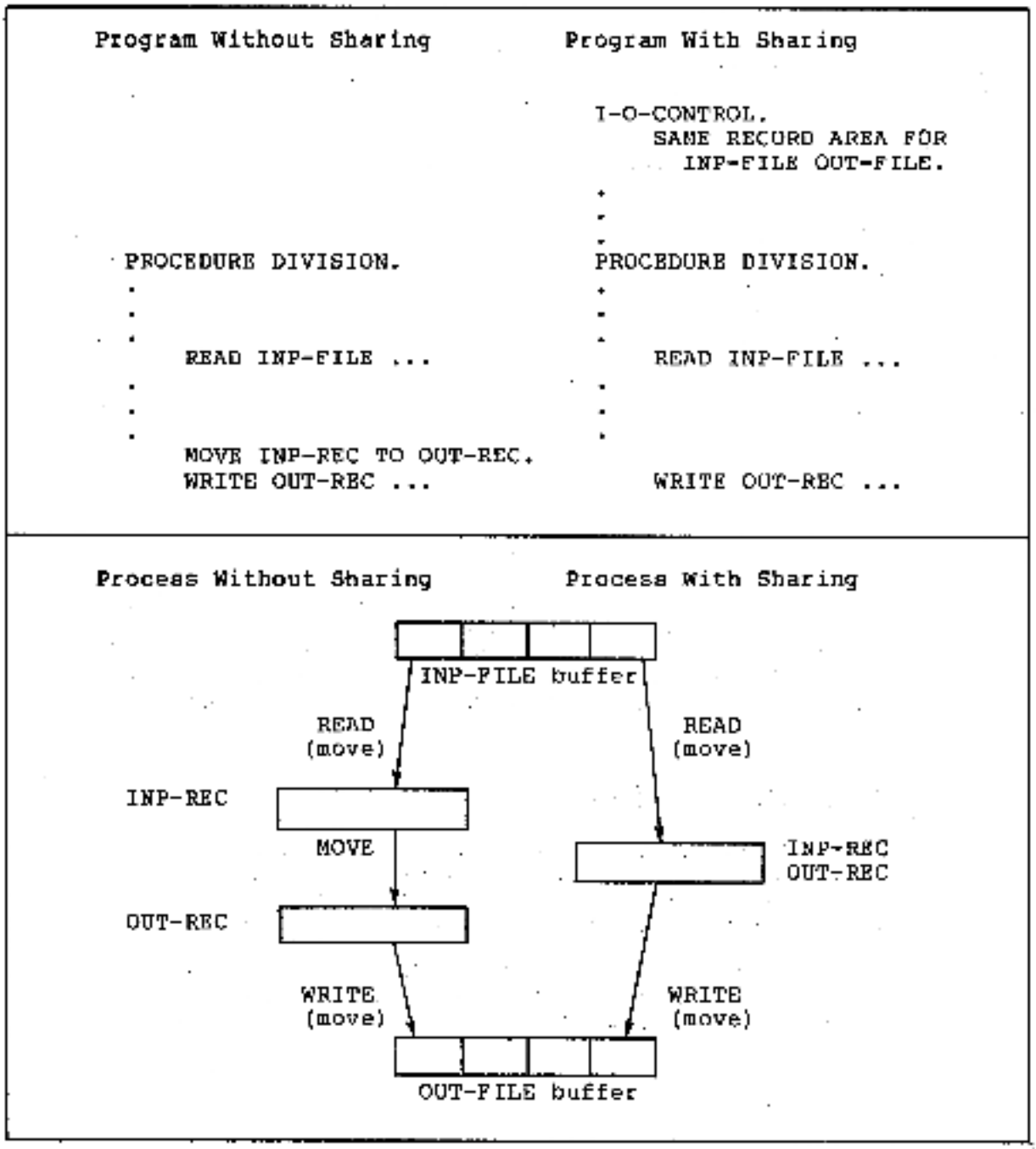
2. Reading records from one file and writing them to another requires an intermediate data transfer from one file's current record area to the other. If the program processes a large number of records this way, the data movement operations could add significant processing overhead.

Files can share current record areas, thus reducing both address space and processing overhead. You specify current record area sharing by using the SAME RECORD AREA clause in the I-O-CONTROL paragraph of the Environment Division. For all files named in a SAME RECORD AREA clause, the compiler assigns the beginning of the current record areas to a common location; the leftmost bytes of each current record area coincide in the same way that the leftmost bytes of each record for a file share one location. Records need not be the same size; nor must the maximum sizes of each current record area be the same.

Figure 6-1 shows the effect of current record area sharing in a program that reads records from one file and writes them to another. However, it also shows a drawback: current record area sharing is

equivalent to implicit redefinition; the records do not exist separately -- therefore, if the program changes the record defined for the output file, the original input file record is no longer available. Remember that you cannot access a file's I/O buffer directly.

Figure 6-1
Sharing Record Areas



6.5 I/O BUFFERS

An I/O buffer is an intermediate memory storage location for data transfers between a program and its files. RMS assigns buffers dynamically; that is, it does not allocate address space for a file's buffers until your program opens the file. Furthermore, when your program closes a file, RMS releases the I/O buffer's address space.

Using buffers, RMS can perform I/O operations with little regard to the program's description of records and files: it can read or write more data (or less) in each operation than the program requests.

Buffer use is necessary for record blocking; for example, when your program reads a record that is part of a logical block, only that record is made available in the current record area. The rest of the records in the logical block are still available in the file's I/O buffer, and RMS can often make them available to your program without accessing the file again.

Multi-buffering (allocating more than one buffer for file operations) can increase the speed of I/O operations. Using multi-buffering, RMS can reduce your program's record access time by storing large amounts of data in the program's address space between I/O requests. If the program tries to access a record that is already in the buffer, RMS must only move the record to the current record area, regardless of how the file is blocked. You can take advantage of multi-buffering by using the RMS defaults or by specifying multiple buffers with the RESERVE clause.

6.5.1 RMS Buffer Defaults

RMS uses default multi-buffer counts when they are not specified by your program. Defaults for sequential, relative, and indexed files can be set on a system-wide basis; however, you can also define RMS defaults for your process by using the SET RMS_DEFAULT command. You can display the defaults with the SHOW RMS_DEFAULTS command. The **VAX/VMS Command Language User's Guide** describes both commands.

6.5.2 Multiple Buffers (RESERVE Clause)

You can use the RESERVE clause in the SELECT statement to specify the number of I/O buffers that RMS will use for a file. The RESERVE clause specification overrides the RMS default, allowing you to reserve more buffers for programs in which I/O speed is important.

You can specify up to 127 areas in the RESERVE clause; however, if record or block sizes are large, heavy multi-buffering could cause the buffers to take a large proportion of the process address space.

6.5.3 Sharing Buffers (SAME AREA Clause)

The SAME AREA clause specifies that two or more files are to use the same memory area (I/O buffers) during processing. It is not valid to have more than one of the files open at the same time; the RTS reports a run-time error when it detects this condition.

RMS allocates I/O buffers dynamically (when your program opens a file), so buffer sharing would not save resources; therefore, since more than one file specified in a SAME AREA clause cannot be open at the same time, buffers are not actually shared. However, you can use the SAME AREA clause to ensure that specific files are closed before others are opened.

6.6 OPENING FILES

A COBOL program must explicitly open a file before it can perform any I/O operation on it. You can open files in four modes: INPUT, OUTPUT, I-O, and EXTEND; the choice of open mode determines which I/O statements you can use. This section summarizes the I/O statements that can be used for each open mode; it also discusses the procedures used by the RTS and RMS when you open files.

6.6.1 I/O Operations

Three conditions determine the I/O operations that a program can perform on a file:

1. File organization
2. Access mode
3. Open mode

The relationships among file organization, access mode, open mode, and I/O statements are hierarchical: file organization determines which access modes are valid; the combination of organization and access mode determines valid open modes; and the combination of all three enables or disables I/O statements. Table 6-1 shows these relationships by indicating I/O statement availability for each valid combination.

Table 6-1
I/O Statements Grouped by File Organization,
Access Mode, and Open Mode

File Organization	Access Mode	Statement	Open Mode			
			INPUT	OUTPUT	I-O	EXTEND
SEQUENTIAL	SEQUENTIAL	READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		WRITE	No	Yes	No	Yes
RELATIVE	SEQUENTIAL	DELETE	No	No	Yes	N/A
		READ	Yes	No	Yes	
		REWRITE	No	No	Yes	
		START	Yes	No	Yes	
		WRITE	No	Yes	No	
	RANDOM	DELETE	No	No	Yes	
		READ	Yes	No	Yes	
		REWRITE	No	No	Yes	
		WRITE	No	Yes	Yes	
	DYNAMIC	DELETE	No	No	Yes	
		READ	Yes	No	Yes	
		READ NEXT	Yes	No	Yes	
		REWRITE	No	No	Yes	
		START	Yes	No	Yes	
		WRITE	No	Yes	Yes	
INDEXED	SEQUENTIAL	DELETE	No	No	Yes	N/A
		READ	Yes	No	Yes	
		REWRITE	No	No	Yes	
		START	Yes	No	Yes	
		WRITE	No	Yes	No	
	RANDOM	DELETE	No	No	Yes	
		READ	Yes	No	Yes	
		REWRITE	No	No	Yes	
		WRITE	No	Yes	Yes	
	DYNAMIC	DELETE	No	No	Yes	
		READ	Yes	No	Yes	
		READ NEXT	Yes	No	Yes	
		REWRITE	No	No	Yes	
		START	Yes	No	Yes	
		WRITE	No	Yes	Yes	

6.6.2 OPEN Statement Execution

This section discusses the file open procedure and how it is affected by the following conditions:

- File organization
- Access mode
- Open mode
- RTS error detection
- RMS error detection
- Existence of USE procedures

An OPEN statement causes the RTS to begin a series of procedures that attempt to make the file available to the program. If an error is detected, the OPEN fails: the RTS either performs the applicable USE procedure (if the program has one) or issues an error message and terminates the image.

The following procedure starts when a COBOL program reaches an OPEN statement:

1. The RTS checks the current status of the file. If it is already open, the OPEN fails.
2. The RTS builds a file specification by using the contents of the VALUE OF ID identifier, if any, to replace or add to the components of the ASSIGN clause file specification default.
3. If the file was named in a SAME AREA clause, the RTS checks the status of all other files named in the clause. If any are open, the OPEN fails.
4. The RTS calls RMS, requesting that it open the file. If RMS detects an error in its procedures, it reports the error condition to the RTS and the OPEN fails.
5. RMS passes the file specification to a logical name translation routine, which replaces the file specification with the translation, if one exists.
6. If the file specification names an invalid device, or if RMS detects any other error in the file specification, it reports the error to the RTS.
7. If RMS cannot find the file, it notifies the RTS; then, if the program specified the file as OPTIONAL (valid for sequential files only) and the open mode is INPUT, the RTS marks the file for an AT END condition, considers the OPEN successful, and returns control to the program. Otherwise, if RMS reports that it cannot find the file, the OPEN fails.

8. If the RTS detects a significant difference between an existing file's attributes and those specified during an open for input (such as different file organization), the OPEN fails.

9. If the program is opening a sequential file for output, and the program contained either the LINAGE or APPLY PRINT-CONTROL clauses, the RTS initializes the LINAGE counters.

10. Before returning control to the program after a successful OPEN, the RTS marks as enabled or disabled all the program's I/O statements that refer to the file, depending on the file organization, access mode, and open mode. (See Table 6-1.)

6.7 NAMING FILES

In a COBOL program, you refer to a file by its file-name: the name you specify in the FD and use in the SELECT, OPEN, READ, START, and CLOSE statements. However, you refer to the physical file, as it exists outside the program, with a file specification. The ASSIGN clause of the SELECT statement contains the default file specification; the VALUE OF ID clause in the file description particularizes the file specification.

This section discusses file specifications and logical names, which are described in detail in the VAX/VMS Command Language User's Guide. It relates the ASSIGN and VALUE OF ID clauses to the two file naming techniques and recommends a convention that you can follow to make your COBOL programs device-independent.

NOTE

The term "system" is used when it is not important to differentiate between VAX/VMS and its subsystems (RMS, for example).

6.7.1 File Specifications

File specifications provide the system with all the information it needs to uniquely identify a file or device.

The general form of the file specification is:

```
node::device:[directory]filename.type;ver
```

The punctuation marks and brackets separate the fields of the file specification. The fields are:

node — A network node name identifies a location on the network, if your system is connected to one. Node names are from one to six characters long.

device — Each hardware device in the system has a unique device name specification in the format:

```
devcu:
```

where dev is a mnemonic for the device type; c is a controller designation; and u is a unit number.

directory — This field names a directory file, which contains the identifications and locations of a user's files on a disk device. Directory names must be enclosed in either square brackets ([and]) or angle brackets (< and >).

filename — This field, in combination with the file type and version number, uniquely identifies files within directories. The file name can be from one to nine characters long.

type — The file type is often used to identify a file in terms of its contents. It can be from one to three characters long. For example, the file type of executable images is usually EXE.

ver — Version numbers are decimal numbers from 1 to 32767 that differentiate between versions of a file. For example, if you create a file with the same file name and type as one that already exists in the same directory, the system assigns a version number that is one greater than the highest existing version. You must put a semicolon or a period before the version number.

All fields are optional in a file specification. The system supplies default values for all omitted fields, except file name and type. For example, default device and directory names are established when you log in; you can change them with the SET DEFAULT command.

File Specification Examples:

```
DBA1:[SMITH]ACCOUNT.DAT
```

Refers to the latest version (default) of the file named ACCOUNT.DAT in directory name [SMITH] on device DBA1: on the local system (node name default).

```
PAYROL.NEW;23
```

Refers to version number 23 of the file named PAYROL.NEW in the default directory on the default device.

File Switches

File specifications in the ASSIGN or VALUE OF ID clause can be followed by "file switches". These specifications are not defined by VAX-11 RMS; however, the RTS accepts them and translates them to VAX-11 RMS parameters. COBOL-74 accepts file switches for compatibility with PDP-11 COBOL. (They are discussed in Section 6.8.5).

6.7.2 Logical Names

Logical names allow you to write programs that are independent of physical file specifications. They also provide a shorthand way to specify files that you refer to frequently.

When the system gets a file specification, it tries to find an equivalence name to replace the leftmost component. If the leftmost file specification component is not a logical name -- it might be a directory name, for example -- the system does not translate it.

You can assign logical names with the ASSIGN command, which is described in the **VAX/VMS Command Language User's Guide.** When a logical name is assigned, it and its equivalence name are placed in one of three logical name tables, depending on whether they are assigned for the current process, on the group level, or on a system-wide basis.

To translate a logical name, the system searches the three tables in order (process, group, system); therefore, you can override a system-wide logical name by defining it for your group or your process.

Logical name translation is a recursive procedure; that is, when the system translates a logical name, it uses the equivalence name as the argument for another logical name translation. It continues in this manner until it cannot translate the equivalence name.

For example, assume that the equivalence name of FILEA is ALPHA.DAT, and you use the ASSIGN command:

```
ASSIGN FILEA MYFILE
```

If your program tries to open a file whose file specification is "MYFILE", the system translates "MYFILE" to its equivalence name, "FILEA", then uses "FILEA" as the argument for a second translation to "ALPHA.DAT". If you had not established "MYFILE" as a logical name, the system would use it as a file specification and look for a file named "MYFILE.".

6.7.3 ASSIGN and VALUE OF ID Clauses

You can use the SELECT statement ASSIGN clause alone or in combination with the VALUE OF ID clause to supply file specifications and logical names for files. The ASSIGN clause literal can contain three types of file names:

1. Complete file specification

   If you want the file specification to be the same whenever you run the program, you can use a complete explicit file specification in the ASSIGN clause. For example:

```
SELECT WORK-FILE
    ASSIGN TO "DBA1:[WORKACCT]WORK1.TMP".
```

   You need not use the VALUE OF ID clause if you choose this technique.

2. Partial file specification

   If part of the file specification can change from one run to another, you can use a partial file specification:

```
SELECT INPUT-FILE
    ASSIGN TO "DBA1:  .DAT".
```

   If you do not use the VALUE OF ID clause, the system uses the default directory and a null file name. However, if you use the VALUE OF ID IS identifier clause and use the ACCEPT statement to get the file name at run time, the contents of the identifier "fill in" or replace all or part of the partial file specification:

identifier contents	file specification
[JONES]A	DBA1:[JONES]A.DAT
PERSONNEL	DBA1:PERSONNEL.DAT
[WILLIAMS]	DBA1:[WILLIAMS].DAT
spaces	DBA1:.DAT
DBA0:[SMITH]WEEK.LIS	DBA0:[SMITH]WEEK.LIS
.TMP	DBA1:.TMP

3. Logical name

   A logical name can look like a partial file specification and, in fact, can be used in the same way. However, if the name is not changed (by the contents of the VALUE OF ID) so that the system recognizes it as a file specification, the system tries to translate it.

For example, assume that the logical name "MYFILE" translates to the equivalence name "PERSONNEL.DAT". If the SELECT statement is:

```
SELECT INP-FILE
    ASSIGN TO "MYFILE".
```

and the program accepts input at run time to fill the VALUE OF ID identifier, logical name translation may or may not take place, depending on the input:

identifier contents	file specification
.DAT	MYFILE.DAT
spaces	PERSONNEL.DAT
[SMITH]	[SMITH]MYFILE.

You can often increase program versatility and avoid "one-time" recompilations by using logical names in the ASSIGN clause literal. Consider an accounting system with 20 programs that access the same ten master files. It could be necessary to use the same programs to access another set of master files at the start of a new year, for example, when two accounting years are active, or when another organization's accounts must also be maintained. The amount of work involved in converting the programs would depend on the programs' file naming conventions:

- If complete file specifications are used in the ASSIGN clause, and the programs do not accept file specifications at run time, all 20 programs need to be edited (ten file specification changes in each), linked, and probably renamed to avoid confusion with the originals. The obvious risk of error requires that the programs be tested.

- If the programs accept file specifications at run time, they would not need to be changed. However, assuming they run as batch jobs, each command procedure needs to be changed, and probably renamed. If the programs run interactively, operators would need to be taught to use the correct file specifications for each program, depending on which set of files was to be used. In addition to being cumbersome, this procedure would likely result in file specification errors, and perhaps damage to data files.

- If logical names are used in the ASSIGN clause, the programs would not need to be changed. If the programs run as batch jobs, the command procedures are simpler than in the previous case, since logical name assignments need be made only once at the beginning of the procedure instead of for each program; if multiple command procedures were used, they could all execute a single command procedure that does nothing but assign logical names. If the programs run interactively, the operator needs only to execute one command procedure to assign all logical names for the correct set of files.

6.7.4 File Switches (PDP-11 COBOL Compatibility)

PDP-11 COBOL allows the use of switches (qualifiers) in file specifications to communicate options to the file management system, RMS-11. The RTS translates these switch specifications to make them compatible with VAX-11 RMS; thus, you can recompile most PDP-11 COBOL programs with VAX-11 COBOL-74 and execute them without change.

Table 6-2 describes the PDP-11 COBOL file specification switches. Note that some of the terminology and concepts are not the same as those in the VAX-11 system. Note also that numeric values are specified as either decimal or octal, not hexadecimal.

Table 6-2
File Specification Switches
for Compatibility with PDP-11 COBOL

SWITCH	MEANING
/AL:n	Allocate n disk blocks to the file when it is created. This ensures that n blocks are available before processing begins. You can also use the switch to ensure that the the volume can hold the entire file. If the rightmost character of n is a decimal point, the RTS interprets the value as a decimal number; otherwise, the RTS treats n as an octal value. The blocks allocated need not be contiguous.
/CL:n	The RTS treats this switch identically to the /WI switch. The /CL switch is included for compatibility with PDP-11 COBOL programs running under the RSTS/E operating system.
/CO:n	This switch complements /AL:n; it further specifies that all blocks be contiguous. If the rightmost character of n is a decimal point, the RTS interprets the value as a decimal number; otherwise, the RTS treats n as an octal value.
/EQ:n	Specifies an extension quantity of n blocks when the file is created. A large extension quantity minimizes extend operations. If the rightmost character of n is a decimal point, the RTS interprets the value as a decimal number; otherwise, the RTS treats n as an octal value.
/DW	Causes I/O buffers to be written only when full, as contrasted to the default case, in which every write operation causes a physical I/O operation. This option is available only for files that are not write-shared.
/LO	Causes RMS to use the fill numbers specified when the file was created. Fill numbers can cause the file to contain free space to allow later record insertion.

Table 6-2 (Continued)

/MI — This switch optimizes the insertion (into an indexed file) of records sorted in order of ascending prime key values. Mass insertion eliminates the index search for subsequent writes. This feature is implemented in RMS-11.

/SH — Specifies sharing of the file, making it available for writing by other processes running concurrently with the COBOL program. This switch is not allowed for sequential files. For other types of files, the following rules apply:

1. If the /SH is specified for one process sharing the file, it must be specified for all processes sharing the file.
2. If a file is being opened for output or I/O with the /SH switch specified, all other processes currently using the file must also have the /SH switch specified.
3. If a file is opened for input without the /SH switch set, no other process can use the file for output or I/O.
4. If a file is opened for input without the /SH switch set, no other process currently using the file can have the /SH switch set.

If access is denied because one of these rules has been violated, the RTS stores a value of 91 in the FILE-STATUS data-item associated with the file, assuming that the SELECT statement for the file contains a FILE-STATUS clause.

/WI:n — Sets the number of retrieval pointers in the window used to map virtual block numbers to logical block numbers. The acceptable values range from 1 to 102 if you know exactly how many pointers are present on disk for the file, or 255, which requests assignment of pointers as needed.

CAUTION

The /WI:n switch can cause loss of data and file integrity if the system crashes while a buffer is being filled.

6.8 FILE COMPATIBILITY

VAX-11 COBOL-74 programs use VAX-11 RMS to access all files. In most cases, therefore, your COBOL programs can read records from any file that was created through (or can be accessed by) RMS; and other RMS-based programs can read records from files that your program creates.

The ability to read records, however, does not imply a universal ability to transfer data between programs written in different languages -- at least not without some special processing. The most common compatibility issues are differences in data types, data record formats, and special control characters in records that are written to record-oriented devices.

6.8.1 Data Type Differences

Not all data types are supported by all programming languages and all utilities. For example:

- COBOL-74 does not support the floating point data type. Therefore, COBOL programs cannot easily process floating point data (FORTRAN real variables) in files.
- VAX-11 SORT does not support quadword binary items. Therefore, problems arise in sorting a file on a COMPUTATIONAL key whose picture is in the range S9(10) to S9(18).
- FORTRAN does not support the packed-decimal data type. Therefore, COMPUTATIONAL-3 data in COBOL files cannot easily be used by FORTRAN programs.

The fact that data types differ should not keep you from transferring data between programs written in different languages. You can use two techniques to overcome data type incompatibilities:

1. Use ASCII character representation, if possible, for all data in files intended for use across languages: ASCII representation (USAGE IS DISPLAY, in COBOL) is almost universally recognized. All VAX-11 languages and utilities support this data type.

2. Convert the data in your program to a data format you can use. Even if you cannot use a data type directly, you can convert it to a usable form if you know the specifications for its internal representation.

   Data conversion may be complex and may prevent your operating on the converted data as easily as on a "native" data type. Nevertheless, conversion is always possible. For example, a FORTRAN double-precision value may have too large a magnitude for representation in any COBOL data type; however, the fraction and exponent can be represented in COBOL by two COMPUTATIONAL data items.

6.8.2 Data Record Formatting Differences

Programming languages may use different conventions to format their data records, causing incompatibilities in otherwise transportable files. For example, FORTRAN programs normally place a carriage control character before the first data character in a formatted file record. Similarly, other languages could format print-controlled (and other) records differently than COBOL does.

You can avoid this incompatibility, in some cases, by not using print-controlled files. In FORTRAN, for example, a file can be opened with the "CARRIAGE CONTROL='NONE'" specification.

If you cannot "normalize" the format of a file's records, you can still read it by defining record descriptions that match the actual format. For example, you can read data from a FORTRAN file that uses carriage control by defining a one-character data item before the first "real" data item in each record description. When you read a record, the one-character field will contain the carriage control character, which your program can either interpret or ignore.

6.8.3 Special Control Characters

Some characters in the computer character set have special meaning; the carriage return, form feed, and line feed characters are examples of non-graphic control characters.

Control characters may be included in a file's records by convention; for example, COBOL-74 print-controlled records have a carriage return in the byte following the last data character. In other cases, a program can inadvertently include control characters in a record by using a data type other than ASCII (or DISPLAY). For example, if your program writes COMPUTATIONAL data to a file, individual bytes of the binary data could contain control characters. Consider a one-word COMPUTATIONAL data item that contains the value 3085, which is equivalent to the hexadecimal value 0C0D; taken as two bytes, 0C0D represents a form feed followed by a carriage return, which could be interpreted as the end of a record.

6.9 I/O ERROR PROCESSING

When your program reaches a file I/O statement, the RTS begins a complex procedure that includes its own internal status checks and interaction with RMS. I/O exceptions can be detected by RMS or the RTS. This section briefly describes how the RTS handles exception conditions and the techniques you can use to handle I/O errors.

When your program reads a sequentially accessed file, RMS reports an AT END condition to the RTS if there are no more records to return. For a randomly accessed file, an INVALID KEY condition is reported whenever RMS determines that the file does not contain the record specified by the value of the key your program supplied. The AT END and INVALID KEY conditions are not errors; you specify program action on those conditions in the AT END and INVALID KEY clauses of I/O statements.

An I/O error is any other condition that causes an I/O statement to fail.

If your program contains a USE procedure that applies to the file for which the I/O operation failed, the RTS performs the procedure, and it does not display an RMS error message; otherwise, it displays the RMS error message and terminates the image.

A USE procedure can sometimes avoid program termination. For example, if the file status data item contains the value "91", which indicates that the file is locked by another process, you might decide to try opening the file again after performing other procedures.

In other cases, when program continuation is not desirable, the USE procedure can perform "housekeeping" operations that conclude processing in an orderly way, by saving data or informing the user, for example.

Before the RTS performs a USE procedure, it places a value in the file status data item, if you specified one in the file's SELECT statement. Most file status values are defined by the 1974 ANSI COBOL Standard; in most cases, they do not provide as much information as RMS error messages. If you need to see the RMS error message for an error that was handled by a USE procedure, you must recompile the program without the USE procedure and run it again.

6.10 LOW-VOLUME I/O (ACCEPT AND DISPLAY)

The COBOL language provides two statements (ACCEPT and DISPLAY) for low-volume I/O operations. Usually, these statements transfer data to and from a user's terminal device. In COBOL-74, however, the ACCEPT and DISPLAY statements refer to VAX/VMS logical names.

This section discusses the association of your own mnemonic-names to VAX/VMS logical names; it continues with discussions of the ACCEPT and DISPLAY statements.

6.10.1 Mnemonic-Names (SPECIAL-NAMES Paragraph)

The ACCEPT and DISPLAY statements transfer data between your program and the object of VAX/VMS logical names. If you do not use the FROM/UPON clauses, the default logical names are COB$INPUT and COB$OUTPUT.

The FROM/UPON clauses refer to mnemonic-names that you can define in the SPECIAL-NAMES paragraph in the Environment Division. You define a mnemonic-name by equating it to a "device"; for example, the following clause equates STATUS-REPORT to the device LINE-PRINTER:

```
LINE-PRINTER IS STATUS-REPORT
```

You could then use the mnemonic-name in a DISPLAY statement:

```
DISPLAY "File contains " REC-COUNT
    UPON STATUS-REPORT.
```

6.10.2 Logical Name "Devices"

The device names in the SPECIAL-NAMES paragraph represent VAX/VMS logical names:

SPECIAL-NAMES Device	Logical Name
CARD-READER	COB$CARDREADER
PAPER-TAPE-READER	COB$PAPERTAPEREADER
CONSOLE	COB$CONSOLE
LINE-PRINTER	COB$LINEPRINTER
PAPER-TAPE-PUNCH	COB$PAPERTAPEPUNCH

The logical names do not necessarily represent devices. You could, for example, assign a logical name to a file specification with a VMS ASSIGN command:

```
ASSIGN [ALLSTATUS]STATUS.LIS COB$LINEPRINTER
```

Because a logical name does not imply a device, it carries no implication of "open mode"; therefore, a program can display upon a mnemonic-name that refers to CARD-READER or accept from a mnemonic-name that refers to LINE-PRINTER.

In COBOL, the ACCEPT and DISPLAY statements do not refer to file-names; therefore, the concepts of opening and closing files do not apply. However, the RTS uses RMS for all I/O operations, including ACCEPT and DISPLAY. The RTS therefore implicitly "opens" a logical name when it is first used in an ACCEPT or DISPLAY statement in any COBOL module in the image.

NOTE

When the RTS opens a logical name for a DISPLAY statement, it specifies the variable with fixed-length control (VFC) format to allow carriage control; the RTS does not use VFC format when it opens a logical name for an ACCEPT statement. The record format attribute is used for all operations until the image terminates; therefore, if your program contains both ACCEPT and DISPLAY statements that refer to the same logical name, it should execute a DISPLAY before the first ACCEPT. Otherwise, DISPLAY statement carriage control will be lost; all DISPLAY statements will execute as if they contained the WITH NO ADVANCING phrase.

This condition does not occur when you use ACCEPT and DISPLAY statements without the FROM/UPON clause: the statements refer to different logical names (COB$INPUT and COB$OUTPUT).

6.10.3 ACCEPT Statement

Format 1 of the ACCEPT statement transfers small amounts of data from the object of a VAX/VMS logical name to a data item. If you do not use the FROM clause, the RTS uses the logical name COB$INPUT; otherwise, it uses the logical name implied by the key word in the SPECIAL-NAMES paragraph that is referred to by the mnemonic-name in the ACCEPT statement. In the following example, the RTS uses COB$CONSOLE:

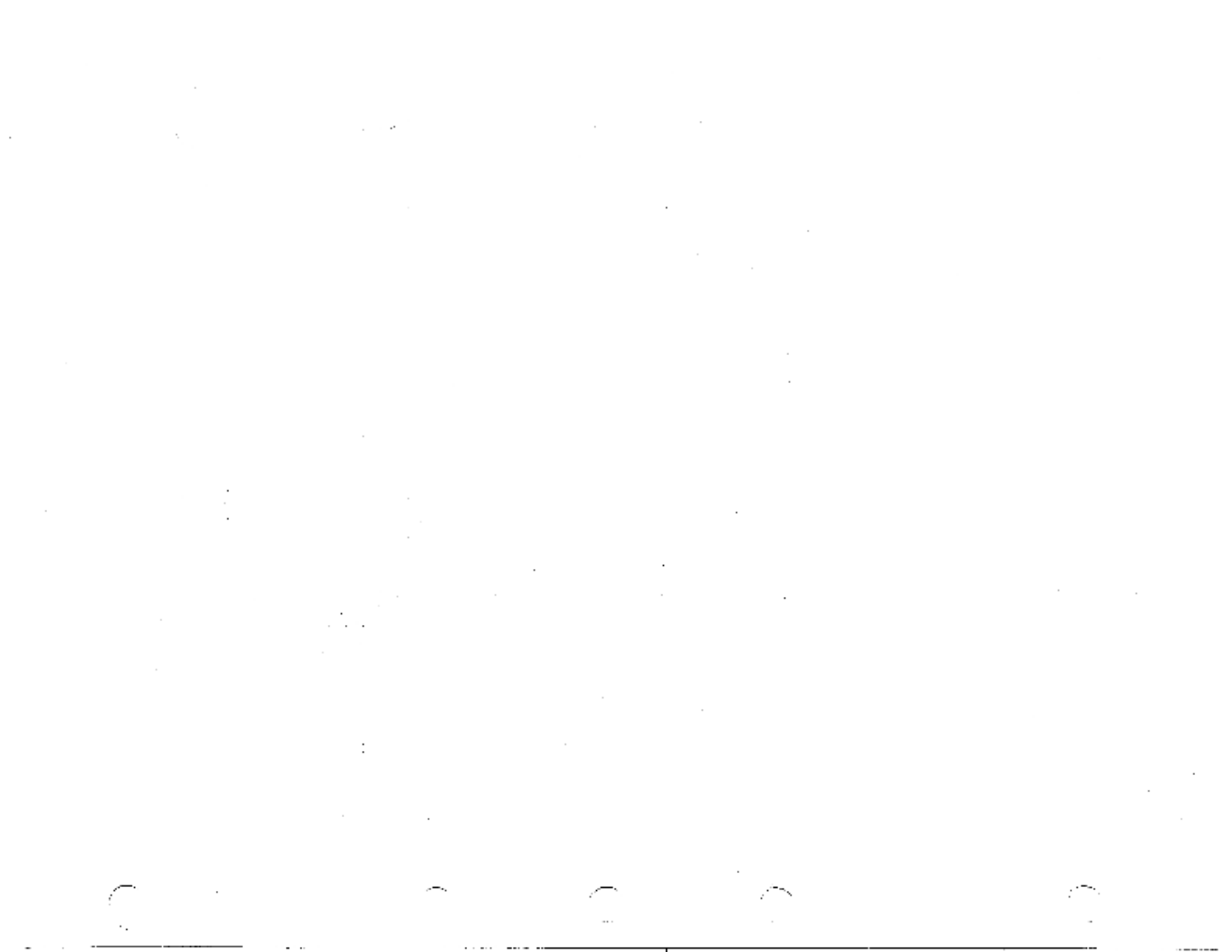
```
SPECIAL-NAMES.
    CONSOLE IS WHATS-HIS-NAME
    .
    .
    .
PROCEDURE DIVISION.
    .
    .
    .
    ACCEPT PARAMETER-AREA FROM WHATS-HIS-NAME.
```

6.10.4 DISPLAY Statement

The DISPLAY statement transfers the contents of data items and literals to the object of a VAX/VMS logical name. If you do not use the UPON clause, the RTS uses the logical name COB$OUTPUT; otherwise, it uses the logical name implied by the key word in the SPECIAL-NAMES paragraph that is referred to by the mnemonic-name in the DISPLAY statement. In the following example, the RTS uses COB$LINEPRINTER:

```
SPECIAL-NAMES.
    LINE-PRINTER IS ERROR-LOG
    .
    .
    .
PROCEDURE DIVISION.
    .
    .
    .
    DISPLAY ERROR-COUNT, " phase 2 errors, ",.
            ERROR-MSG UPON ERROR-LOG.
```

For the DISPLAY statement, the RTS uses the variable with fixed-length control record format.



CHAPTER 7

GOOD PROGRAMMING PRACTICES

7.1 FORMATTING THE SOURCE PROGRAM

Since most COBOL programs are usually long, the programmer needs techniques that will help him to simplify and improve the readability of his COBOL programs. The guidelines in this chapter, if followed, will help produce source programs that are easy to read and maintain.

Before considering these guidelines, consider the reference formats that are available with VAX-11 COBOL-74:

1. The Conventional (ANS) format.

2. The Terminal format.

Although the Conventional format produces ANS compatible programs, it also produces source printouts that are somewhat more cluttered than those produced by the Terminal format. These guidelines, therefore, recommend the use of Terminal format and all of the following suggestions and examples assume the use of that format. Besides the obvious advantage of an uncluttered printout, the Terminal format has other programming advantages:

1. It requires less storage area.

2. It requires no line numbers.

3. Its statements may be aligned with tab characters.

Further, whenever required, the REFORMAT utility program will convert Terminal format programs to the Conventional format. (The REFORMAT utility program is discussed in Chapter 8).

The following suggestions should help to further simplify even the most complicated source programs.

1. Begin division, section, and paragraph names in column 1. Although these names may start anywhere in Area A, aligning them in column 1 produces a much more readable listing. When required, place the * and - in column 1. (Column 1 then becomes column 0.)

2. Insert a blank line, or one or more comment lines (describing the purpose of the file) before each SELECT statement in the FILE-CONTROL paragraph. Place the phrases of the SELECT statement on separate lines and begin each of them in column 5 (use the tab character to skip over Area A). Consider the following illustration of a typical SELECT statement:

```
AREA A          AREA B

1 . . .         5 . . . . . .
                SELECT MASTER-FILE
                ASSIGN TO "DB1:"
                ORGANIZATION IS RELATIVE
                ACCESS IS SEQUENTIAL.
```

3. Place the phrases of the file description statement on separate lines and begin each of them in column 5. (Use the tab to skip over Area A.) Consider the following illustration of a typical file description entry:

```
AREA A          AREA B

1 . . .         5 . . . . . .
FD              MASTER-FILE
                LABEL RECORDS ARE STANDARD
                VALUE OF ID IS MASTER-FILE-NAME
                DATA RECORD IS MASTER-RECORD.
```

4. In both the File and Working-Storage sections, begin all 01 level items in column 1.

   Indent, by four columns, all subordinate items with higher-valued level numbers. (For example, if the item that is subordinate to a 01-level record description is 05, begin the record description level number in column 1 and the 05 level number in column 5.) Use the tab character for the first indentation, a tab and four spaces for the second, two tabs for the third, etc. When indented in this manner, the listing will show, clearly and neatly, the hierarchical relationships of all of the data names in the program as well as their level number values.

   Increment level numbers by 5; then later, if it becomes necessary to insert additional group items, they may be inserted without having to change the level numbers of all items that are subordinate to that group.

If desired, write the level numbers as single digits (such as 1 instead of 01).

Use level number 01 instead of 77 in the Working-Storage Section. (77, as a level number has the same meaning as 01, and 77 may eventually be omitted from the COBOL standard.)

Since all elementary items, except for index data items, require PICTURE clauses, these clauses fill a good part of the source program listing. However, the PICTURE clause itself may be simplified to enhance the listing's readability as follows:

a. Use PIC as an abbreviation for PICTURE.

b. Omit the noiseword IS.

c. Align the PIC clauses on successive lines. (Use the tab character to align the clauses.)

5. Put all paragraph name declarations in the Procedure Division on lines separate from the statements in the paragraph. This not only makes the program more readable, it also makes modification of the first statement in the paragraph easier.

6. Follow all imperative statements with a period, making them 1-statement sentences. Place only one statement on a line. In addition to making the lines shorter and more readable, this will prove quite helpful when debugging the program. For example, if the program contains a coding error, it will be on one line and therefore easier to modify without affecting the other portions of the sentence; further, the diagnostic messages will refer to the correct line and their meanings will be clearer.

   Since left-aligned statements in any program enhance the readability of that program, develop the habit of starting all COBOL sentences in column 5. (Use the tab character to skip over Area A.) Some statements, however, should be further indented, as explained in the following paragraphs.

7. If the true path of a conditional statement contains another conditional statement or more than one imperative statement, place all statements in the true path on lines immediately following the conditional statement and indent them to show their dependence upon that statement. Consider the following illustration of an IF statement and its true path:

```
IF COMPUTED-TAX > TAX-LIMIT
   SUBTRACT TAX-LIMIT FROM COMPUTED-TAX GIVING EXCESS-TAX
   MOVE TAX-LIMIT TO COMPUTED-TAX
   ADD EXCESS-TAX TO TOTAL-EXCESS-TAX.
```

   If the statement has an ELSE (or false) path, align the word ELSE under the preceding IF and indent all statements that are dependent on the ELSE statement.

Thus:

```
IF condition
   true path statement
   true path statement
   . . .
ELSE
   false path statement
   false path statement.
```

Be sure to place the period after the last statement only!

Another good method for simplifying conditionals is to write only a single imperative statement in the true or false path. If the path requires more statements, place them in a separate paragraph and either PERFORM the paragraph from the path or GO to it. This technique avoids the possibility of inadvertently placing a period at the end of a statement within the path, thereby terminating it prematurely.

When writing a GO TO ... DEPENDING statement, place each procedure name on a separate line and indent them all. Consider the readability of the following sample statement:

```
GO TO P35
      P40
      P45
      P60
      P65
      DEPENDING ON P-SWITCH.
```

8. When grouping statements into paragraphs and sections, use the following organizational ideas:

   Group together logical units of processing into a section. Select a section name that reflects the type of processing being conducted within that section (such as TAX-COMPUTATION SECTION, PRINT-LINE-FORMATTER SECTION, etc.). Follow the section name with sufficient comment lines to explain the processing that is carried out by the statements within that section.

   Make paragraph names as short and simple as possible. A numbered abbreviation of the section name often suffices. Thus the paragraph names in the TAX-COMPUTATION section might be TC10, TC20, TC30, etc. Use paragraph names sparingly, placing them only where the true and false paths of conditional statements require branch points for GO TO statements. If the temptation arises to give a paragraph a longer name in an attempt to reflect the type of processing in that paragraph, use comment lines instead. (Comment lines usually convey more information, more clearly.)

   When using simple numbered paragraph names, assign increasing numeric characters to sequential paragraphs. If the numeric portion of the names increases by 5 or 10, new ones may be inserted later without disturbing the sequence of the names.

Do not use the PERFORM verb in the form, PERFORM a THRU b. If the paragraphs a thru b must be performed, place them in a section by themselves and PERFORM the section, thus avoiding the use of the THRU option.

Place single paragraphs that are to be performed into sections and use the section name as the object of PERFORM verbs. Then, if future design changes introduce complicated conditional logic into the paragraph, requiring additional paragraph names, the PERFORM statements need not be altered.

The preceding guidelines divide the Procedure Division into modular blocks of coding. If those guidelines are used, the following additional techniques may be applied.

a. Restrict entry to all sections through the first statement of the section by use of a GO TO, a PERFORM, or a "fall through" from the preceding section.

b. Ensure that all GO TO statements refer to only section names or paragraph names that are internal to the section containing the GO TO statement.

7.2 USE OF PUNCTUATION

Avoid using the COBOL punctuation characters, comma and semicolon. They lend little to the readability of programs that have their statements neatly aligned, as discussed earlier in this chapter. Further, it is quite easy to misuse these characters, which can cause serious errors for many compilers. (Other compilers either ignore incorrect punctuation characters or flag them with warning messages.) At best, even when used correctly and in the proper places, they have no effect on the meaning of the program.

7.3 USE OF THE ALTER STATEMENT

Avoid using the ALTER statement to change the flow of control in a program. It is impossible to test the setting of an alterable GO statement except by executing it. Also, unless explicit comments accompany an alterable GO statement, it is difficult to tell whether or not it is referenced by ALTER statements or what the possible destinations might be. All of this makes debugging programs that contain these statements quite difficult. There are two other techniques that may be used in their place:

1. If control branches one of two ways (i.e., a binary switch), write the switch as a conditional variable. Consider the following sample coding:

```
01  P-SWITCH PIC S9 COMP VALUE 0.
    88 NO-PRINT VALUE 1.

    MOVE 1 TO P-SWITCH
    .
    .
    .
    IF NO-PRINT GO TO P40.

P40.
    MOVE 0  TO P-SWITCH.
```

2. If control branches more than two ways, use MOVE statements to place integers into a data item, and a GO TO ... DEPENDING ... statement to test the data item and branch accordingly. Consider the following sample coding:

```
01  P-SWITCH PIC S9999 COMP VALUE 0.
    .
    .
    .
    MOVE 1 TO P-SWITCH
    .
    .
    .
    MOVE 3 TO P-SWITCH.
    .
    .
    .
    GO TO
         PART-TIME
         PIECE-WORK
         HOURLY
         SALARIED-WEEKLY
         SALARIED-OTHER
         DEPENDING ON P-SWITCH.
*   FALLTHROUGH IS A BUG
    DISPLAY "?17".
    STOP RUN.
```

7.4 USE OF THE PERFORM STATEMENT

The general rules for the PERFORM statement are augmented with the following rules:

1. The endpoint of a section and the endpoint of the last paragraph in the same section are two distinct points. This means that it is possible to execute a PERFORM of the section, then while that PERFORM is still active, to execute a PERFORM of the last paragraph.

2. On the start of a PERFORM, if the end point of the new PERFORM is the end point of an already active PERFORM, the RTS aborts the task and issues an error message.

3. At the end of any procedure, a check is made to see if the procedure being ended is the end of the most recent PERFORM range. If so, the most recent PERFORM range is exited. If not, the end point of the most recent procedure is checked against the end point of all currently active PERFORMs. If the end point of the procedure is the end point of any currently active PERFORM range, the RTS issues an error message and aborts the task because the perform ranges are not being exited in the reverse of the order in which they were entered.

NOTE

The RTS error messages are discussed in Chapter 10.

7.5 USE OF LEVEL-88 CONDITION-NAMES

Condition-names provide a convenient method for testing a value or range of values in a field. The use of condition-names makes programs easier to maintain, because it ensures a uniform method of testing fields and helps to reduce recoding when the specifications of the program change.

The following example illustrates the use of condition-names and shows the advantages inherent in their use.

Suppose the records of a file each describe a student in an educational institution (or an employee in a corporation). Some of the records contain categories of information which are not present in other records. A "code" field, which contains a digit or letter, indicates the presence (or type) of some categories; while a special value in the information itself (such as a numeric value being zero, negative, or maximum) indicates the presence of other categories. The processing of such a record may vary considerably depending on these indicator fields. The fields may require interrogation at various points in the program, and the interrogation may require more than a simple relation test.

Consider a "code" field that holds one of seven values, coded as a mnemonic character. For example, S,1,2,3,4,G,P might be seven values that indicate student categories of Special, 1st year, 2nd year, 3rd year, 4th year, Graduate, and Postgraduate. The field is described as follows:

```
05 STUDENT-CATEGORY PIC X.
```

Program logic requires certain processing for enrolled undergraduates, different processing for special students, and still different processing for all students except enrolled undergraduates.

Without the aid of condition-names, statements might be written as follows to resolve this problem:

```
IF STUDENT-CATEGORY = "S" ...

IF STUDENT-CATEGORY NOT LESS THAN "1"
       IF STUDENT-CATEGORY NOT GREATER THAN "4" ...

IF STUDENT-CATEGORY EQUAL TO "G" NEXT SENTENCE
       ELSE IF STUDENT-CATEGORY EQUAL TO "P"
            NEXT SENTENCE ELSE GO TO ...
```

However, if various level 88 entries follow the STUDENT-CATEGORY description, as shown below, condition-names can simplify this coding.

```
05 STUDENT-CATEGORY PIC X.
   88 UNDERGRADUATE VALUE "1" THRU "4".
   88 SPECIAL-STUDENT VALUE "S".
   88 GRAD-STUDENT VALUE "G" "P".
   88 SENIOR VALUE "4".
   88 NON-DEGREE-STUDENT VALUE "S" "P".
```

Now, the following procedural statements can solve the problem:

```
IF SPECIAL-STUDENT ...
IF UNDERGRADUATE ...
IF GRAD-STUDENT ...
```

Procedural statements with condition-names are much easier to read and debug than those containing the complete test. For example, the procedural statements, IF UNDERGRADUATE ..., and IF STUDENT-CATEGORY NOT LESS THAN "1" IF STUDENT-CATEGORY NOT GREATER THAN "4" both accomplish the same thing, but the first statement is simpler and less confusing.

In addition, the statement, IF NOT UNDERGRADUATE ... can test the category of not being an undergraduate, which is equivalent to any one of the following statements:

```
IF NOT (STUDENT-CATEGORY NOT < "1" AND
       STUDENT-CATEGORY NOT > "4") ...

                or

IF STUDENT-CATEGORY < "1" OR
   STUDENT-CATEGORY > "4" ...

                or

IF STUDENT-CATEGORY < "1" NEXT SENTENCE
   ELSE IF STUDENT-CATEGORY > "4" NEXT SENTENCE
       ELSE GO TO ...
```

Statements such as these are tedious to write and a frequent source of coding errors. Further, if a change creates a new student category, the recoding takes more time and is even more error prone.

A careful and controlled use of condition-names forces a higher degree of programming control and checkout. If the program logic does require the modification of the STUDENT-CATEGORY field, it can even be named FILLER thus removing the opportunity to shortcut the use of condition-names.

To apply condition-names, follow the description of the item to be tested with a level 88 entry. The item being tested, known as the conditional variable (STUDENT-CATEGORY in the preceding illustrations), may be either DISPLAY or COMPUTATIONAL usage, but not INDEX usage; it may also be a group item.

The compiler stores all of the values supplied by the level 88 entries in the object program exactly as written. (They are pooled with all of the literals from the Procedure Division.) A value supplied by a level 88 entry for a conditional variable of COMPUTATIONAL usage is stored in binary format to save conversion at run time. The compiler stores all other values as byte strings with the proper attributes. It does not make the level 88 entries equal to their conditional-variables in size. This means that it neither truncates nor pads (with spaces) non-numeric literals. Further, it neither truncates nor pads (with zeros) numeric literals, but stores them as written or, if converted to binary, in the minimum size COMP item that will hold the converted value. It stores signs as trailing overpunches on numeric DISPLAY literals, and removes and remembers decimal points.

Do not enter level 88 items under group items that have subordinate entries containing any of the following clauses: SYNCHRONIZED, JUSTIFIED, COMPUTATIONAL, INDEX.

7.6 USE OF QUALIFIED REFERENCES

7.6.1 Qualified Data References

The COBOL language provides facilities to define and reference user-defined data items. Data items are programmer-defined variables declared in the Data Division of a COBOL program. Such variables include, among others, file record descriptions and internal working areas. These data items are processed by procedural statements such as the WRITE, MOVE, and ADD statements. Procedural operations on these data are facilitated through references to the data items by name.

For example, to update a variable, YTD-GROSS-PAY, by a weekly gross pay amount WEEKLY-GROSS, write the program fragment shown in Figure 7-1.

```
.
.
.
WORKING-STORAGE SECTION.
01 YTD-GROSS-PAY  PIC 9(5)V99.
01 WEEKLY-GROSS   PIC 999V99.
.
.
.
   ADD WEEKLY-GROSS TO YTD-GROSS-PAY.
```

Figure 7-1
Unqualified Data Item Reference

In this example, YTD-GROSS-PAY and WEEKLY-GROSS are defined in the Working Storage Section of the Data Division as COBOL variables with a level number of 01. The variable representing the "year-to-date gross pay (YTD-GROSS-PAY)" is computed by incrementing its present value by the "weekly gross pay (WEEKLY-GROSS)" amount through reference to the appropriate data items in the ADD statement. References are made to the data items by the singular, unqualified names of YTD-GROSS-PAY and WEEKLY-GROSS. Since YTD-GROSS-PAY and WEEKLY-GROSS are defined with level numbers of 01 in the Working Storage Section, these variables must be unique in their spelling and, hence, can only be referenced by the spelling of each data item's name without any COBOL qualification.

The example in Figure 7-1 is artificial because the data item representing the "year-to-date gross pay" is defined as a level 1 variable in the Working Storage Section. More realistically, YTD-GROSS-PAY is defined as a field within an employee payroll record residing on an external master payroll file. The process of updating the "year-to-date gross pay" by a "weekly gross pay" amount is shown more appropriately in Figure 7-2.

```
.
.
.
FILE SECTION.
FD MASTER-IN
   LABEL RECORD IS STANDARD
   VALUE OF ID IS "MASTER.PAY".
01 PAY-RECORD.
   03 NAME              PIC X(30).
   03 EMPLOYEE-NO       PIC 9(9).
   03 YTD-GROSS-PAY     PIC 9(5)V99.
.
.
.
```

```
FD MASTER-OUT
   LABEL RECORD IS STANDARD
   VALUE OF ID IS "MASTER.PAY".
01 PAY-RECORD.
   03 NAME              PIC X(30).
   03 EMPLOYEE-NO       PIC 9(9).
   03 YTD-GROSS-PAY     PIC 9(5)V99.
.
.
.
WORKING-STORAGE SECTION.
01 WEEKLY-GROSS         PIC 999V99.
.
.
.
PROCEDURE DIVISION.
INIT.
   OPEN INPUT MASTER-IN.
   OPEN OUTPUT MASTER-OUT.
.
.
.
   ADD WEEKLY-GROSS, YTD-GROSS-PAY OF MASTER-IN
               GIVING YTD-GROSS-PAY OF MASTER-OUT.
.
.
.
```

Figure 7-2
Qualified Data Item Reference

In this example, YTD-GROSS-PAY is defined as a field in both the input and output record descriptions. There are two separate data items whose spellings are identical.

To reference each data item, it is necessary to qualify the name of each data item with sufficient information to constitute a unique reference. Thus, to reference the "year-to-date gross pay" amount in the output record, we write "YTD-GROSS-PAY OF MASTER-OUT" where such a reference is called a qualified reference. The filename MASTER-OUT is functioning as a qualifier in the reference. The reserved word "OF" is the qualification connector and may be used interchangeabely with the reserved word "IN" in this context. Another way of referencing the same data item is to write "YTD-GROSS-PAY OF PAY-RECORD IN MASTER-OUT". This reference is called a completely qualified reference because all possible qualifiers are specified in the reference. A reference of the form "YTD-GROSS-PAY" or "YTD-GROSS-PAY OF PAY-RECORD" is illegal since it does not uniquely identify which of the two data items is desired. Such a reference is termed an ambiguous reference.

In the area of data item definition and referencing, COBOL is unlike other languages such as FORTRAN and ALGOL 60. While FORTRAN requires each data item to have a unique name (i.e., no two data items may have a name of identical spelling), COBOL relaxes this requirement to the

extent that each data item must be uniquely referable. That is, two or more data items may have their names spelled identically, but there must exist a way to reference each distinct data item. Thus, there is a distinction between a data item and its name. Central to understanding this distinction is understanding the concept of unique referability.

The functionalities of data item definition and referencing may be understood by stating three guidelines which relate the concepts of data item definition, reference format, and unique referability.

7.6.2 Guideline 1 (Data Item Definition)

Each data item has a name. Each name is immediately preceded by an associated positive integer called its level number. A name either refers to an elementary item or else it is the name of a group of one or more items whose names follow. In the latter case, each item in the group must have the same level number, which must be greater than the level number of the group item.

7.6.3 Guideline 2 (Reference Format)

Data-name qualification is performed by following a data-name or condition-name by one or more phrases of a qualifier preceded by IN or OF. IN and OF are logically equivalent. The general format of a qualified reference to an elementary item or group of items named "name-0" is given in Figure 7-3.

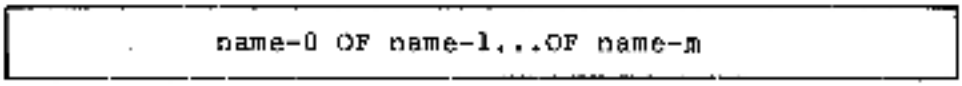


Figure 7-3
General Format of a Qualified Data Reference

where m >= 0 and where, for 0 <= j < m, name-j is the name of some item contained directly or indirectly within a group item named "name-j+1". A reference of the form given in Figure 7-3 is called a (partially) qualified reference with name-1,name-2,...,name-m being called qualifiers. Such a reference is termed a completely qualified reference if "name-j+1" is the father of name-j for 0 <= j <= m-1.

In the hierarchy of qualification, names associated with an FD indicator are the most significant, then the names associated with level-number 01, then names associated with level-number 02,...,49. The most significant name in the hierarchy must be unique and cannot be qualified. Subscripted or indexed data-names, unsubscripted data-names, and condition variables may be made unique by qualification. The name of a condition variable can be used as a qualifier for any of its condition-names.

Enough qualification must be mentioned to make the reference unique; however, it may not be necessary to mention all levels of the hierarchy as the example in Figure 7-2 demonstrates.

7.6.4 Guideline 3 (Unique Referability)

If more than one data item is defined with the same name "name-0", there must be a way to refer to each use of the name by using qualification. That is, each definition of "name-0" must be uniquely referable. A data item is uniquely referable if the complete set of qualifiers for the data item are not identical to any partial (including complete) set of qualifiers for another data item.

7.6.5 Qualified Procedure Statements

The facility of qualification may be applied to procedure references. A procedure name is either a paragraph or section name. By definition, a paragraph name is unique only within a section containing the paragraph while, on the other hand, section names must be unique within a COBOL program. The general format of a qualified procedure reference is shown in Figure 7-4.

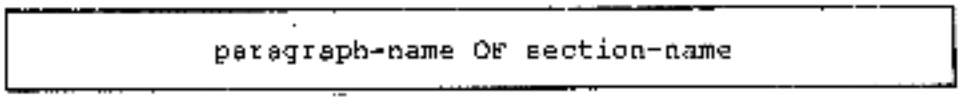


Figure 7-4
General Format of a Qualified Procedure Reference

A paragraph name may be qualified by its containing section name; a section name may never be qualified in a procedure reference. When a paragraph name is referenced without an explicit section name qualifier, the paragraph name is implicitly qualified by the appropriate section name.

If a paragraph name is unique within a COBOL program it is not necessary to qualify the paragraph name in the procedure reference. Finally, if a paragraph name is not unique within a COBOL program, the paragraph name must be qualified in a procedure reference when the reference is made outside of the section which contains the paragraph.

7.6.6 Qualification and Compiler Performance

Qualification is a powerful language facility for the development of COBOL programs. Used wisely, it increases the readability of COBOL programs. However, the user pays a price for utilization of this facility in terms of a slower compilation rate (i.e., COBOL source lines per unit of time).

Qualification requires a tree-structured symbol table at compile-time. The time required for building and looking up on a tree-structured symbol table is considerably longer than for a non-tree-structured symbol table. This translates into a general degradation of compiler performance. If qualification is not employed in a program compiled by the VAX-11 COBOL-74 compiler, compilation speed is not affected. However, when qualification is used, the compilation rate slows down due to the additional system overhead.

In general, if there are deeper levels of qualification, there will be a slower compilation. This is especially so at the end of the Data Division text where duplicate data-name declarations are detected by the compiler. Run-time performance is not affected by usage of the qualification facility.

CHAPTER 8

REFORMAT UTILITY PROGRAM

VAX-11 COBOL-74 accepts source programs that were coded using either the conventional 80-column card reference format or the shorter, terminal-oriented VAX-11 cobol terminal format. The REFORMAT utility program reads source programs that were coded in the terminal format and converts them to 80-column conventional format source programs. The **VAX-11 COBOL Language Reference Manual** discusses both formats in detail.

Consider the two formats:

- The terminal format is designed for ease of use with text editors controlled from an on-line console keyboard and is compatible for use with the VAX-11 system. It eliminates the line-number and identification fields and allows horizontal tab characters and short lines.

- The conventional format produces source programs that are compatible with the reference format of other COBOL compilers throughout the industry.

REFORMAT lets you write source programs in the terminal format; then, if compatibility is ever required for any of those programs, it provides a simple method for conversion to the conventional format.

REFORMAT follows the following steps to expand each line of terminal format coding to the conventional format:

- It generates a 6-character line number of 000010, places that number in the first six character positions of the line, and increases it by 000010 for each subsequent line.

- It places any continuation or comment symbols (-,*, or /) into character position 7.

- It places the coding from the terminal format line into character positions 8-72, thereby creating a line of conventional format coding.

- It replaces any horizontal tabs with the appropriate number of space characters to simulate tab stops at character positions 5, 13, 21, 29, 37, 45, 53, 61, and 66 of the terminal format line.

- It moves spaces into any character positions left between the last character of coding and character position 73.

- It places either identification characters (if they were supplied at program initialization) or spaces into character positions 73-80.

- It right justifies (at position 72) the first line of a continued non-numeric literal, thus guaranteeing that the literal will remain the same length as it was in the default format.

- It right justifies (at position 72) the first part of any COBOL word that is split over two lines.

- It creates a line containing a slash (/) in position 7 and space characters in positions 8 through 72 for every form-feed character that it encounters.

REFORMAT Command String

To run REFORMAT, enter the following command:

```
MCR RFM
```

This causes REFORMAT to begin execution. REFORMAT immediately requests the file specifications for the two files (input and output) to be processed. In response to its prompting messages, type in the file specifications for your two files.

```
RFM-INPUT FILE SPEC:
RFM-OUTPUT FILE SPEC:
```

When the system has successfully opened both files, REFORMAT types the following request for an identification entry in columns 73 through 80. If you desire an identification entry, type in from one to eight characters. REFORMAT places these characters, left justified, in columns 73 through 80 of each output line. If no entry is required, type a carriage return.

```
RFM-COLS 73 TO 80:
```

Following this response, REFORMAT reads the input file and writes it as 80-character records, in conventional reference format.

When it has processed the last record in the file, REFORMAT displays the following messages; the first indicating the number (nnnnn) of output records produced and the second requesting another input file.

```
RFM-nnnnn LINES PROCESSED.
RFM-INPUT FILE SPEC:
```

If there is another file to be reformatted, follow the same sequence with the specifications for the next file. If not, type CTRL/Z to terminate execution.

REFORMAT Error Messages

If any of the responses to the prompting messages contain detectable errors, REFORMAT displays the following messages indicating the problem.

```
RFM-ERROR IN OPENING INPUT FILE
RFM-TRY AGAIN
RFM-INPUT FILE SPEC:
```

The system could not open the input file. Either the file is not present on the device specified (the default device is SYS$DISK) or the file name is typed incorrectly. The usual I/O error messages precede this message.

To continue processing that file, examine the input file spec and type in a corrected version. To process another file, type in a new input file specification. To terminate execution, type CTRL/Z.

```
RFM-ERROR IN OPENING OUTPUT FILE
RFM-TRY AGAIN
RFM-OUTPUT FILE SPEC:
```

The system could not open the output file. An incorrectly typed file specification usually causes this error. (The default device is SYS$DISK.) The usual I/O error messages precede this message. To continue, examine the output file specification and type in a corrected version. To terminate execution, type CTRL/Z.

```
RFM-INPUT FILE IS EMPTY
RFM-INPUT FILE SPEC:
```

The system successfully opened the input file, but the first READ statement encountered the AT END condition.

To continue, type in a new input file specification for another file. To terminate execution, type CTRL/Z.

```
RFM-ERROR IN READING INPUT FILE
RFM-INPUT FILE SPEC:
```

The first attempt to read the input file was unsuccessful. This error is usually caused by an input record length exceeding 86 characters. (Although terminal format records should not exceed 66 characters in length, REFORMAT provides a record area of 86 characters and ignores the right-most 20 characters.)

To continue, type in a new input file specification for another file. To terminate execution, type CTRL/Z.

```
RFM-ERROR IN READING INPUT FILE
RFM-REFORMATTING ABORTED
RFM-nnnnn LINES PROCESSED
RFM-INPUT FILE SPEC:
```

While reading input records (other than the first record), REFORMAT was unsuccessful in an attempt to read a record. It terminates execution and closes both files.

To process another file, type in a new input file specification and continue with the prompting message sequence. To terminate execution, type CTRL/Z.

```
RFM-ERROR IN WRITING OUTPUT FILE
RFM-REFORMATTING ABORTED
RFM-nnnnn LINES PROCESSED
RFM-INPUT FILE SPEC:
```

REFORMAT was unsuccessful in an attempt to write an output record. It terminates execution and closes both files.

To process another file, type in a new input file specification and continue with the prompting message sequence. To terminate execution, type CTRL/Z.

CHAPTER 9

DEBUGGING COBOL PROGRAMS

Debugging is the process of finding and correcting errors in programs that have been successfully compiled and linked. In some cases, you need to debug a program because it produces incorrect results; other programs may terminate abnormally as a result of attempting illegal operations.

This chapter introduces the VAX-11 Symbolic Debugger (DEBUG) for COBOL-74 programs. You will find a complete description of the DEBUG facility in the VAX-11 Symbolic Debugger Reference Manual; enough information is included here to get you started debugging a COBOL program.

9.1 DEBUG CONCEPTS

DEBUG is a module that you can include in an executable image with the LINK command. (See Chapter 2.) It allows you to examine and change the contents of your program's data items; you can also control the order of statement execution and regain control when execution errors occur.

The VAX-11 Symbolic Debugger is called symbolic because it allows you to refer to data items by the symbols you assigned, that is, data-names. You can refer to Procedure Division locations by source listing line numbers rather than VAX-11 memory addresses.

To use symbolic references, the compiler must store the symbols in the object module. Although this adds no address space requirement to the executable image, it does require space in the image file; that is why symbol tables are not stored automatically -- you must cause them to be stored by using the /DEBUG command qualifier when you compile a program. If you do not specify /DEBUG at compile time, you can still use the debugger, but you cannot refer to data items symbolically.

To summarize, use the /DEBUG command qualifier at both compile time and link time. Then, when you execute the image, DEBUG takes control and prompts you for a command with the prompt: DBG>.

9.2 PREPARING TO DEBUG A PROGRAM

The following sections describe the commands that establish the environment for debugging COBOL programs. The commands are:

```
SET LANGUAGE COBOL

SET MODULE
SHOW MODULE
CANCEL MODULE

SET SCOPE
SHOW SCOPE
CANCEL SCOPE
```

9.2.1 SET LANGUAGE COBOL Command

This command tells DEBUG that the debugging dialog applies to a COBOL program. It allows symbols that contain hyphens, for example. The format of the command is:

```
SET LANGUAGE COBOL
```

You may want to debug an image that contains modules written in more than one language; the SET LANGUAGE command allows you to change language conventions during the debugging session.

DEBUG's default is the language of the main program.

9.2.2 MODULE Commands: SET, SHOW, and CANCEL

DEBUG maintains a table of symbols defined in the program with which it is linked; the table contains the name of each data item defined in the program, its data type, and its location. The table can hold about 2000 symbols at a time. Therefore, if an image contains modules that have a total of more than 2000 symbols, the table may not be able to hold all of them at once. You cannot refer to a symbol unless it is in the active symbol table.

Use the MODULE commands to control the contents of DEBUG's active symbol table when the image you want to debug contains multiple modules. The commands are:

SET MODULE — Places the symbols defined in the specified module (program) into the active symbol table.

The format of the SET MODULE command is:

```
SET MODULE module-name [,module-name] ...
```

SHOW MODULE — Displays the names of the modules whose symbols are currently in the active symbol table.

The format of the SHOW MODULE command is:

```
SHOW MODULE
```

DEBUG responds by displaying the names of the modules linked with it; it also indicates which modules' symbols are in the active symbol table and how much space they occupy.

CANCEL MODULE — Removes a module's symbols from the active symbol table.

The format of the CANCEL MODULE command is:

```
CANCEL MODULE module-name [,module-name] ...
```

9.2.3 SCOPE Commands: SET, SHOW, and CANCEL

The SCOPE commands control the default that DEBUG uses to resolve references to symbols. When you use the EXAMINE command, for example, you can either name the module in which the symbol is defined, or you can omit the module name. If you omit the name, DEBUG uses a default; if it can't find the symbol in the default scope, it attempts to find an unambiguous symbol in the remaining modules. If DEBUG cannot resolve the reference, it displays a message.

Until you use a SET SCOPE command, DEBUG uses as the default scope the name of the first module with which it was linked.

The SCOPE commands are:

SET SCOPE — Specifies the default module.

The format of the SET SCOPE command is:

```
SET SCOPE module-name
```

SHOW SCOPE — Displays the current default module name.

The format of the SHOW SCOPE command is:

```
SHOW SCOPE
```

CANCEL SCOPE — Cancels the current default module name. Until you use another SET SCOPE command, DEBUG uses the name of the first module with which it was linked as the default.

The format of the CANCEL SCOPE command is:

```
CANCEL SCOPE
```

9.3 SPECIFYING LOCATIONS

Several DEBUG commands use locations as parameters; locations allow you to tell DEBUG what data you want to look at or where you want control transferred, for example.

9.3.1 Location Types

For COBOL programs, you can use three types of location specifications:

Data-name — refers to a data item in the Data Division. This type of location is often called a symbol.

For example:

```
EXAMINE INPCHAR
```

tells DEBUG that you want to see the contents of the data item whose data-name is INPCHAR.

You cannot qualify data-names in DEBUG commands the way you can in COBOL. If you refer to a data-name that is defined more than once in the module, DEBUG applies the reference to the definition that appeared last in the source program.

You can use subscripts in DEBUG commands as in COBOL statements, except for data-names appearing in the Linkage Section; however, you can usually access Linkage Section items by referring to the corresponding non-Linkage Section data-name in the calling program.

Line — specifies the beginning of a source program line. The format of a line location is:

```
%LINE n
```

The value of n corresponds to a compiler-assigned line number on the program's source listing.

Absolute — specifies a numeric memory address. Specify an absolute location as an integer.

For example:

```
EXAMINE 1200
```

tells DEBUG to display the contents of the longword located at address 1200.

9.3.2 Resolving Location Ambiguities

Your program can consist of more than one module, as it would if a main program called a subprogram. If the symbols from more than one module are in DEBUG's symbol table, and if duplicates exist, then a symbol reference could be ambiguous. Furthermore, if more than one module were linked with DEBUG, then line numbers could also be ambiguous.

DEBUG uses defaults to resolve ambiguous references. In some cases, however, you may want to specify the scope in an location to refer to other than the default module.

For example, your program might consist of a main program, TESTA, and a subprogram, TESTB. If you wanted to transfer control to line 36 in TESTB, you might use the command:

```
GO %LINE 36
```

Because you did not specify a module name, DEBUG uses the default scope to resolve the ambiguity -- the default could be either TESTA or TESTB, depending on whether you had previously used a SET SCOPE command.

You can override the default and resolve the ambiguity yourself by specifying the scope as part of the location:

```
GO %LINE TESTB\36
```

Similarly, you can specify the scope in symbolic data-name locations:

```
EXAMINE TESTB\ACCNT-NUM
```

9.4 CONTROLLING PROGRAM EXECUTION

This section describes DEBUG commands that allow you to suspend, monitor, and resume program execution at specific points. The commands are:

SET BREAK	SET WATCH	EXIT
SHOW BREAK	SHOW WATCH	SHOW CALLS
CANCEL BREAK	CANCEL WATCH	
SET TRACE	GO	
SHOW TRACE	STEP	
CANCEL TRACE	CTRL/Y	

9.4.1 BREAK Commands: SET, SHOW, and CANCEL

The BREAK commands control the location of breakpoints in the program. A breakpoint is a location where you want a program to suspend execution and return control to you; at a breakpoint, you can examine or change data values, or you can change the program's execution path.

The BREAK commands are:

SET BREAK — Specifies a location at which to suspend execution.

The format of the SET BREAK command is:

```
SET BREAK location [DO (DEBUG commands)]
```

If the program reaches the specified location, it is suspended before executing the instruction located there. You can request that DEBUG perform commands when the breakpoint is reached by using the DO option. For example:

```
SET BREAK %LINE 210 DO(EXAMINE TOT-AMT)
```

causes DEBUG to display the contents of the data-name TOT-AMT whenever it suspends the program at line 210.

SHOW BREAK — Displays all breakpoints currently set in the program.

The format of the SHOW BREAK command is:

```
SHOW BREAK
```

CANCEL BREAK — Removes specified breakpoints.

The format of the CANCEL BREAK command is:

```
CANCEL BREAK/ALL
```

or

```
CANCEL BREAK location [,location] ...
```

9.4.2 TRACE Commands: SET, SHOW, and CANCEL

The TRACE commands control the location of tracepoints in the program. A tracepoint resembles a breakpoint, except that program execution continues after DEBUG displays the current location. Tracepoints allow you to monitor the sequential flow of a program.

Tracepoints and breakpoints supersede each other; that is, if you set a tracepoint at the same location as a breakpoint, the breakpoint is cancelled.

The TRACE commands are:

SET TRACE — Specifies a location at which to suspend execution, display location information, and continue.

The format of the SET TRACE command is:

```
SET TRACE location
```

SHOW TRACE — Displays the program locations at which tracepoints are currently set.

The format of the SHOW TRACE command is:

```
SHOW TRACE
```

DEBUG displays tracepoints in newest-to-oldest order -- newest is the last tracepoint set.

CANCEL TRACE — Removes specified tracepoints.

The format of the CANCEL TRACE command is:

```
CANCEL TRACE/ALL
```

or

```
CANCEL TRACE location [,location] ...
```

9.4.3 WATCH Commands: SET, SHOW, and CANCEL

The WATCH commands allow you to monitor program locations, called watchpoints, for attempts to change their contents. If an instruction attempts to change the contents of a watchpoint, DEBUG suspends the program, displays the location of the instruction, and prompts for a command. Watchpoints are useful when you need to know if a data item is being inadvertently changed.

The WATCH commands are:

SET WATCH — Specifies locations to be monitored.

The format of the SET WATCH command is:

```
SET WATCH identifier
```

The identifier specifies the location to be monitored.

NOTE

When a watchpoint is set, DEBUG protects the entire memory page from write access. When an instruction at user mode level (your program) attempts to change the contents of any location on the protected page, DEBUG evaluates the access for watchpoint action; however, if a system service tries to write to a protected page, it returns an error. Therefore, if watchpoints are set on the same page as a File Section record description, access errors can occur during RMS input operations.

SHOW WATCH — Displays current watchpoints.

The format of the SHOW WATCH command is:

```
SHOW WATCH
```

DEBUG displays the current watchpoints in newest-to-oldest order -- newest is the last watchpoint set.

CANCEL WATCH — Removes specified watchpoints.

The format of the CANCEL WATCH command is:

```
CANCEL WATCH/ALL
```

or

```
CANCEL WATCH identifier
```

9.4.4 GO and STEP Commands

The GO and STEP commands initiate and continue program execution.

GO — Resumes program execution, either at the current location or another location.

The format of the GO command is:

```
GO [location]
```

If you omit the location, execution starts at the current location.

If you specify an location, DEBUG transfers control to the new location.

You can specify the location as a source program line number (GO %line 38, for example); however, you cannot resume execution at a line boundary if the current location is other than the beginning of a line. If the current location is not a line boundary -- a common occurrence when watchpoints are reached -- use the STEP command to reach the next line boundary before attempting a GO %line command.

NOTE

Exit from DEBUG before restarting a program from the beginning. The results of using a GO command to restart a program from the beginning are undefined.

STEP

Continues execution at the current location for a specified number of steps.

The format of the STEP command is:

```
STEP [n]
```

The value of n specifies the number of steps to execute. If you do not specify n, or you specify 0, DEBUG assumes a value of 1 as a default.

DEBUG evaluates n for each step in the execution of a STEP command. Therefore, if n has a large value, your program runs slower because of DEBUG overhead. You can reduce this overhead by using the SET BREAK and GO commands (instead of STEP) when you want to execute more than a few steps.

9.4.5 CTRL/Y Command (Interrupting the Image)

You can use the CTRL/Y command at any time to return to the VAX/VMS system command level. Press the CTRL key and the Y key at the same time; VAX/VMS displays the $ prompt at the terminal; a STOP literal statement in your program produces the same result. You can then return to DEBUG with the DEBUG command.

Use the CTRL/Y command when you believe your program is in an infinite loop, or when you want immediate control. When you return to DEBUG, you can use the SHOW CALLS command to see the program's location when the CTRL/Y command interrupted execution.

9.4.6 EXIT Command

The EXIT command terminates the debugging session.

The format of the EXIT command is:

```
EXIT
```

DEBUG terminates the program and returns control to the VAX/VMS system command level.

9.4.7 SHOW CALLS Command

The SHOW CALLS command displays information about the current level of nested calls, including performs. The content and format of the information are similar to the traceback display, which is described in Chapter 10.

The format of the SHOW CALLS command is:

```
SHOW CALLS [n]
```

If you do not specify n, DEBUG displays all call levels; otherwise, n determines the number of levels that DEBUG reports.

9.5 EXAMINING AND CHANGING DATA

This section describes commands that allow you to see and to change the contents of data items during program execution. You may want to change values to correct errors or to test a hypothesis during a debugging session. The commands are:

```
EXAMINE

DEPOSIT
```

9.5.1 EXAMINE Command

The EXAMINE command displays the contents of a specified location.

The format of the EXAMINE command is:

```
EXAMINE [location]
```

You will usually specify the location as a data-name (EXAMINE SUB1, for example). However, you can display the contents of an absolute address (such as EXAMINE 1000).

9.5.2 DEPOSIT Command

The DEPOSIT command changes the contents of a specified location.

The format of the DEPOSIT command is:

```
DEPOSIT location=value
```

Examples:

```
DEPOSIT ITEMA=12
```

Places the numeric value 12 into the data item named by data-name ITEMA.

```
DEPOSIT WORDX="NOW IS THE TIME"
```

Places the characters in the alphanumeric literal into the data item named by the data-name WORDX.

```
DEPOSIT TOP="662K"
```

Places the value -6622 into the four-digit signed numeric DISPLAY data item, TOP.

NOTE

The DEPOSIT command functions in the same way as the COBOL ACCEPT statement for DISPLAY data items. Therefore, you must be aware of the internal representation of your program's data items when you use the DEPOSIT command.

9.6 SAMPLE DEBUG SESSION

This section contains an annotated debugging session that demonstrates the use of many DEBUG features. Following it are sample listings of a COBOL program (TESTA) and a subprogram (TESTB); the debugging session refers to these listings.

Program TESTA accepts a character string from the terminal and passes it to TESTB. TESTB reverses the character string and returns it (and its length) to TESTA.

The following debugging session does not demonstrate the location of actual program errors; it is designed to show the use of DEBUG features.

Responses from DEBUG and VMS appear in red.

1) We use the RUN command to start the session. Note that we do not need the /DEBUG qualifier, since the programs were compiled and linked with DEBUG. DEBUG takes control; it displays its standard header, showing us that the default language is COBOL and that the default scope and module are TESTA. DEBUG returns control by displaying its prompt, DBG>.

```
$ RUN TESTA

        VAX/VMS DEBUG V1.5  04 January 1979

%DEBUG-I-INITIAL, language is COBOL, scope and module set to 'TESTA'
DBG>
```

2) Using the SHOW MODULE command, we see that DEBUG's active symbol table contains symbols from only one module, the main program. The last module is part of the RTS; it was linked from C74LIB.OLB.

```
DBG>SHOW MODULE
module name     symbols   language   size

TESTA           yes       COBOL       164
TESTB           no        COBOL       316
CBFLSW          no        BLISS       128

total modules: 3
remaining size: 60948.
```

3) We try to set a breakpoint at line 26 of TESTB. DEBUG cannot find line 26, because TESTB's symbols are not in the active symbol table.

```
DBG>SET BREAK %LINE TESTB\26
%DEBUG-W-NOLINE, routine 'TESTB' has no %line 26
```

4) We add TESTB's symbols with the SET MODULE command.

```
DBG>SET MODULE TESTB
```

5) Then, we confirm that the symbols have been added.

```
DBG>SHOW MODULE
module name     symbols   language   size

TESTA           yes       COBOL       164
TESTB           yes       COBOL       316
CBFLSW          no        BLISS       128

total modules: 3.
remaining size: 60724.
```

6) Now, we can set the breakpoint with no problem.

```
DBG>SET BREAK %LINE TESTB\26
```

7) We resume execution. DEBUG displays the execution starting point. The image continues until TESTA displays its prompt and waits for a response.

```
DBG>GO
routine start at TESTA\TESTA
ENTER WORD
```

8) We enter the word to be reversed. Execution continues until the image reaches the breakpoint at line 26 of module TESTB.

```
BACKWARD
break at TESTB\TESTB %line 26
```

9) We set two breakpoints. When line 40 of TESTB is reached, DEBUG will execute the commands in parentheses; it will display two data items, then resume execution.

```
DBG>SET BREAK %LINE 40 DO(EX HOLD-WORD;EX SUB-1;GO)
DBG>SET BREAK %LINE 34
```

10) We display the active breakpoints.

```
DBG>SHOW BREAK
breakpoint at TESTB\TESTB %line 34
breakpoint at TESTB\TESTB %line 40 DO (EX HOLD-WORD;EX SUB-1;GO)
breakpoint at TESTB\TESTB %line 26
```

11) We set a tracepoint at line 22 of TESTA. TESTA is the default scope.

```
DBG>SET TRACE %LINE 22
```

12) We set a watchpoint on the data-item DISP-COUNT. When the an instruction attempts to change the contents of DISP-COUNT, DEBUG will return control to us.

```
DBG>SET WATCH DISP-COUNT
```

13) We resume execution. Whenever line 40 in TESTB is about to be executed, DEBUG executes the contents of the DO command that we entered at step 9; it displays the contents of HOLD-WORD and SUB-1, then resumes execution.

```
DBG>GO
start at TESTB\TESTB %line 26
break at TESTB\TESTB %line 40
TESTB\TESTB\HOLD-WORD(1:20):  D
TESTB\TESTB\SUB-1(1:2):  8
start at TESTB\TESTB %line 40
break at TESTB\TESTB %line 40
TESTB\TESTB\HOLD-WORD(1:20):  DR
TESTB\TESTB\SUB-1(1:2):  7
start at TESTB\TESTB %line 40
```

```
break at TESTB\TESTB %line 40
TESTB\TESTB\HOLD-WORD(1:20):  DRA
TESTB\TESTB\SUB-1(1:2):  6
start at TESTB\TESTB %line 40
break at TESTB\TESTB %line 40
TESTB\TESTB\HOLD-WORD(1:20):  DRAW
TESTB\TESTB\SUB-1(1:2):  5
start at TESTB\TESTB %line 40
break at TESTB\TESTB %line 40
TESTB\TESTB\HOLD-WORD(1:20):  DRAWK
TESTB\TESTB\SUB-1(1:2):  4
start at TESTB\TESTB %line 40
break at TESTB\TESTB %line 40
TESTB\TESTB\HOLD-WORD(1:20):  DRAWKC
TESTB\TESTB\SUB-1(1:2):  3
start at TESTB\TESTB %line 40
break at TESTB\TESTB %line 40
TESTB\TESTB\HOLD-WORD(1:20):  DRAWKCA
TESTB\TESTB\SUB-1(1:2):  2
start at TESTB\TESTB %line 40
break at TESTB\TESTB %line 40
TESTB\TESTB\HOLD-WORD(1:20):  DRAWKCAB
TESTB\TESTB\SUB-1(1:2):  1
start at TESTB\TESTB %line 40
break at TESTB\TESTB %line 34
```

14) We deposit the value 10 into the data item SUB-1. Note that we do not need to enclose the value in quotes since SUB-1 is a COMP item.

```
DBG>DEPOSIT SUB-1=10
```

15) We examine the contents of SUB-1. DEBUG indicates that it is displaying the value contained in bytes 1 through 2 (1:2) of the data item.

```
DBG>EXAMINE SUB-1
TESTB\TESTB\SUB-1(1:2):  10
```

16) Here, we deposit another value into SUB-1, using a quoted string. DEBUG deposits the value in the same manner as a COBOL ACCEPT statement: as a stream of bytes.

```
DBG>DEPOSIT SUB-1=" "
```

17) When we look at SUB-1, we see that it now has 32 as its value; 32 is the decimal value of the ASCII space character.

```
DBG>EXAMINE SUB-1
TESTB\TESTB\SUB-1(1:2):  32
```

18) We deposit a value into data item SUB-2, whose usage is COMP-3. The quoted string is needed because usage is other than COMP.

```
DBG>DEPOSIT SUB-2="-42"
```

19) We then examine SUB-2, and see that its value is now -42.

```
DBG>EXAMINE SUB-2
TESTB\TESTB\SUB-2(1:2):  -00000000000000000000000000000042
```

20) We look at CHARCT, whose picture is 99V99. DEBUG displays the contents as 0800; it is not aware of the implied decimal point. DEBUG treats all DISPLAY data items as alphanumeric.

```
DBG>EXAMINE CHARCT
TESTB\TESTB\CHARCT(1:4):  0800
```

21) We deposit four characters into CHARCT.

```
DBG>DEPOSIT CHARCT="1500"
```

22) CHARCT now has the value "1500" (15.00).

```
DBG>EXAMINE CHARCT
TESTB\TESTB\CHARCT(1:4):  1500
```

23) Here, we deposit another value, omitting the quotes.

```
DBG>DEPOSIT CHARCT=42
```

24) We examine CHARCT and see that it contains an asterisk (decimal value 42) followed by three spaces.

```
DBG>EXAMINE CHARCT
TESTB\TESTB\CHARCT(1:4):  *
```

25) We deposit the quoted value "15" into CHARCT.

```
DBG>DEPOSIT CHARCT="15"
```

26) Since a quoted string is deposited without conversion, we see that CHARCT now contains "15 ". If we left this value in CHARCT (invalid for a numeric data item), an error would occur later in the run.

```
DBG>EXAMINE CHARCT
TESTB\TESTB\CHARCT(1:4): 15
```

27) So, we deposit a valid value.

```
DBG>DEPOSIT CHARCT="0800"
```

28) We resume execution. The program TESTA displays the reversed word. When the image reaches line 22 in TESTA, DEBUG detects that an instruction has changed the contents of DISP-COUNT. Since we set a watchpoint on DISP-COUNT, DEBUG displays the old and new values, then returns control to us. Note that we don't know the current location in terms of line number; the displayed location is in the RTS.

```
DBG>GO
start at TESTB\TESTB %line 34
trace at TESTA\TESTA %line 22
DRAWKCAB
write to TESTA\TESTA\DISP-COUNT(1:2) at PC CVT_P_ANY+67
        old value =
        new value = 08
```

29) To see the image's current location, we try the SHOW CALLS command. DEBUG displays the active call frames, but we still don't know the line number in our program.

```
DBG>SHOW CALLS
module name     routine name     line    relative PC  absolute PC

                CVT_P_ANY                0000004F     00001414
                TESTA              22    0000007C     00004748
```

30) We use the STEP command until we reach a line boundary. DEBUG indicates that the image has reached line 28 of TESTA, the line following the reference to DISP-COUNT.

```
DBG>STEP
start at CVT_P_ANY+79
stepped to TESTA\TESTA %line 28
```

31) We resume execution. TESTA executes its final DISPLAY. DEBUG regains control when the STOP RUN is executed.

```
DBG>GO
start at TESTATESTA %line 28
08 CHARACTERS
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
```

32) At this point, we can continue the session, by examining the contents of data items, for example; or, we can terminate the image with the EXIT command.

```
DBG>EXIT
$
```

Program Listings:

```
  00001         IDENTIFICATION DIVISION.
  00002         PROGRAM-ID.    TESTA.
  00003         DATE-WRITTEN.  JANUARY 1979.
  00004         DATE-COMPILED.
  00005             15-Jan-1979 .
  00006         ENVIRONMENT DIVISION.
  00007         CONFIGURATION SECTION.
  00008         SOURCE-COMPUTER.     VAX-11.
  00009         OBJECT-COMPUTER.     VAX-11.
  00010         DATA DIVISION.
  00011         WORKING-STORAGE SECTION.
  00012         01  LET-CNT  PIC 9(2)V9(2).
  00013         01  IN-WORD  PIC X(20).
  00014         01  DISP-COUNT PIC 9(2).
  00015         PROCEDURE DIVISION.
  00016         GETIT SECTION.
  00017         BEGINIT.
  00018             DISPLAY "ENTER WORD".
  00019             MOVE SPACES TO IN-WORD.
  00020             ACCEPT IN-WORD.
  00021             CALL "TESTB" USING IN-WORD LET-CNT.
  00022             PERFORM DISPLAYIT.
  00023             STOP RUN.
  00024         DISPLAYIT SECTION.
  00025         SHOW-IT.
  00026             DISPLAY IN-WORD.
  00027             MOVE LET-CNT TO DISP-COUNT.

I 00027 0372  POSSIBLE LOW ORDER RECEIVING FIELD TRUNCATION.

  00028             DISPLAY DISP-COUNT " CHARACTERS".
```

```
00001       IDENTIFICATION DIVISION.
00002       PROGRAM-ID.    TESTB.
00003       DATE-WRITTEN.  JANUARY 1979.
00004       DATE-COMPILED.
00005           15-Jan-1979 .
00006       ENVIRONMENT DIVISION.
00007       CONFIGURATION SECTION.
00008       SOURCE-COMPUTER.    VAX-11.
00009       OBJECT-COMPUTER.    VAX-11.
00010       DATA DIVISION.
00011       WORKING-STORAGE SECTION.
00012       01  SUB-1  PIC 9(2) COMP.
00013       01  SUB-2  PIC 59(2) COMP-3.
00014       01  HOLD-WORD.
00015           03  HOLD-CHAR PIC X OCCURS 20 TIMES.
00016       LINKAGE SECTION.
00017       01  TEMP-WORD.
00018           03  TEMP-CHAR  PIC X OCCURS 20 TIMES.
00019       01  CHARCT PIC 99V99.
00020       PROCEDURE DIVISION USING TEMP-WORD, CHARCT.
00021       CONVERT-IT SECTION.
00022       STARTUP.
00023           IF TEMP-WORD = SPACES
00024             MOVE 0 TO CHARCT
00025             GO TO GET-OUT.
00026           PERFORM LOOK-BACK
00027             VARYING SUB-1 FROM 20 BY -1
00028             UNTIL TEMP-CHAR (SUB-1) NOT = SPACE.
00029           MOVE SUB-1 TO CHARCT.
00030           MOVE SPACES TO HOLD-WORD.
00031           PERFORM MOVE-IT
00032             VARYING SUB-2 FROM 1 BY 1
00033             UNTIL SUB-1 = 0.
00034           MOVE HOLD-WORD TO TEMP-WORD.
00035       GET-OUT.
00036           EXIT PROGRAM.
00037       MOVE-IT.
00038           MOVE TEMP-CHAR (SUB-1)
00039             TO HOLD-CHAR (SUB-2).
00040           SUBTRACT 1 FROM SUB-1.
00041       LOOK-BACK.
00042           EXIT.
```

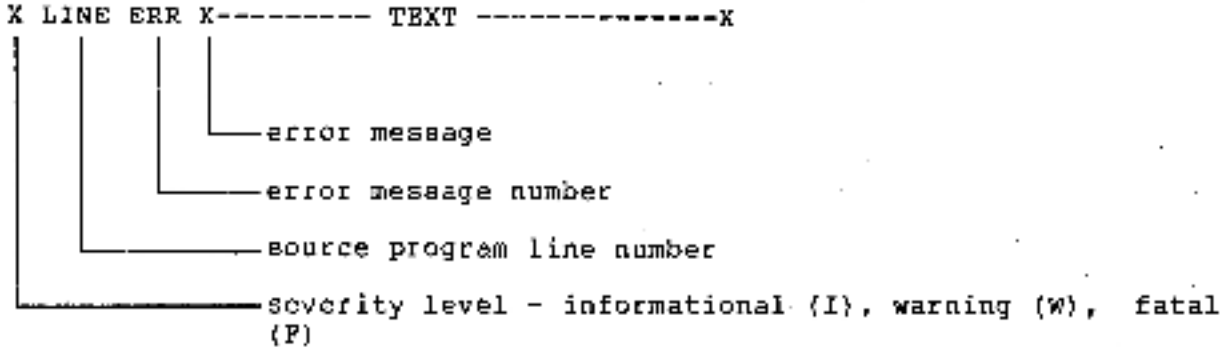
CHAPTER 10

ERROR MESSAGES

Errors can occur while compiling, linking, or running a VAX-11 COBOL-74 program. This chapter defines these different errors and briefly discusses their associated diagnostic messages. Appendix D lists compile-time diagnostics; Appendix E lists run-time error messages.

10.1 COMPILE-TIME ERROR MESSAGES

If the VAX-11 COBOL-74 compiler detects an error, it will print a diagnostic message either before or after the erroneous source program line. Diagnostic messages look like this:



The following example shows diagnostic messages in the source listing of a program named "SAMPLE":

```
  00001          IDENTIFICATION DIVISION.
  00002          PROGRAM-ID. SAMPLE.
  00003          ENVIRONMENT DIVISION.
  00004          SOURCE-COMPUTER. VAX-11.
```

```
  00005          OBJECT-COMPUTER. VAX-11.
  00006          DATA DIVISION.
  00007          WORKING-STORAGE SECTION.
  00008          01 TABLE-VALUES.
  00009               05  TAB-VAL OCCURS 9 TIMES PIC X.
  00010          01 SAVE-VAL                PIC X.
  00011          PROCEDURE DIVISION.
  00012          MAIN SECTION.
  00013          PARA

I 00014  0622    TERMINATOR MISSING AFTER PROCEDURE NAME.

  00014                    IF SAVE-VAL = SPACE
  00015                        ADD 1 TO SAVE-VAL.
       ^
F 00015 0714    MISSING OR INVALID OPERAND FOR ARITHMETIC VERB


I 00016 0616    PROCESSING RESTARTS AFTER TERMINATOR.

  00016               MOVE TAB-VAL (9) TO SAVE-VAL.
  00017               STOP RUN.
  00018               EXIT.

W 00018 0103    .EXIT. WAS NOT THE ONLY VERB IN PARAGRAPH.
```

10.1.1 Severity Levels

Some errors do not affect program compilation, while others abort it. The compiler therefore issues three types of diagnostic messages to reflect varying severity levels: Informational (I), Warning (W), and Fatal (F). Consider the following messages, taken from the SAMPLE program in Section 10.1:

```
I 00014 0622 TERMINATOR MISSING AFTER PROCEDURE NAME
```

The compiler issues informational messages to pinpoint suspect conditions in your source program. In program SAMPLE, the paragraph name PARA does not end with a period. The compiler displays the message: "TERMINATOR MISSING AFTER PROCEDURE NAME." to describe the error. Because the compiler can recover from the error in a manner consistent with your intentions, it issues an informational message only.

NOTE

You can use the /NOWARNINGS command line qualifier to suppress informational error messages.

```
W 00018 0103 .EXIT.  WAS NOT THE ONLY VERB IN PARAGRAPH.
```

Warnings, like informational errors, pinpoint source program mistakes from which the compiler can recover. In program SAMPLE, for instance, the warning indicates that EXIT must be the only statement in the paragraph. The compiler can take corrective action, however. But this action may not be consistent with your intentions, even though the code produced will be executable. The compiler therefore flags the object file to denote that warnings occurred.

```
F 00015 450 REFERENCE TO UNDEFINED DATANAME.  IGNORED.
```

The compiler cannot recover from fatal errors in a manner that reflects your intentions. On line 15 of program SAMPLE, SAVE-VAL is defined as alphanumeric; therefore, it cannot be used in an arithmetic statement. The compiler cannot take corrective action, so it issues a fatal error and does not create an object file. However, it analyzes all remaining source program lines and reports errors.

NOTE

You cannot use a command line qualifier to suppress warning or fatal messages -- they are always printed.

10.1.2 Error Message Printing

The compiler displays the diagnostic error message either before or after the erroneous source program line. There are two exceptions to this rule:

1. Diagnostic messages can appear after the last entry in the DATA DIVISION before the PROCEDURE DIVISION header. These messages reflect errors the compiler cannot report until it has processed the entire DATA DIVISION text.

2. Diagnostic messages can appear after the last line of the PROCEDURE DIVISION. These are messages that the compiler cannot issue until it has processed the entire PROCEDURE DIVISION.

10.1.3 Internal Compiler Errors -- System Errors

The compiler performs consistency checks on program flow and the contents of data fields.

If the compiler detects an inconsistency, it prints a message and terminates compilation. The system error message format is:

```
C74--<error message>
C74--SYSTEM ERROR NNNNNN
```

The six-digit system error code represents the probable cause of the error. When a system error occurs, the compiler closes its input file and does not generate an object file. Appendix F lists system error codes and their meanings.

If an I/O error occurs during compilation, and the compiler cannot continue processing, an I/O system error message is displayed; the compiler then terminates. The format of the I/O system error message is:

```
C74--<error message>
C74--IO ERROR -NN
```

The number (-NN) is an RSX-11M Application Migration Executive (VAX/AME) code.

10.2 SYSTEM MESSAGES

VAX/VMS provides a centralized error message facility. When you type a command at your terminal or execute an image, and an error results, the system displays an error message. The general format for error messages is:

```
%FACILITY-L-CODE, TEXT
[-FACILITY-L-CODE, TEXT]
```

where:

FACILITY	is a VAX/VMS facility, or component name. For example, %C74 represents COBOL-74.
L	is a severity level indicator; it has one of the following values:

Level	Meaning
S	Success
I	Information
W	Warning
E	Error
F	Fatal, or severe error

CODE	is an abbreviation of the message text.
TEXT	is a descriptive message.

If VAX/VMS displays more than one message for an error, the additional message takes the form "-FACILITY-L-CODE, TEXT".

You will find a full discussion of system messages in the **VAX/VMS System Messages and Recovery Procedures Manual.** The following sections discuss system messages issued for link-time or run-time errors.

10.2.1 Link-time Error Messages

The object modules produced by the compiler are nonexecutable; they must first be linked. Two kinds of link-time error messages occur: (1) warning error messages, imbedded in the object module by the compiler (see Section 10.1.1), and (2) errors detected by system facilities invoked by the linker.

If the compiler flags an object file as having warnings, the linker detects the flag and issues the following diagnostics:

```
%LINK-I-WRNERS, MODULE <name> has compilation warnings
%LINK-W-DIAGSISUED, Completed but with diagnostics
```

If a system facility error occurs (for example, if an error occurs when the linker invokes the RMS facility), it is put on a message stack and the linker displays it. Consider the following example. If you typed the command line:

```
LINK XXXX
```

and no object file existed with the name XXXX, the following messages would appear at your terminal:

```
$ LINK XXXX
%LINK-W-OPIDERR, PASS 0 failed to open file "DB1:[ACCOUNT]XXXX.OBJ;"
%LINK-W-UNMCOD, Initial file name was "XXXX", RMS error code = <code>
%RMS-F-FNF, file not found
-SYSTEM-W-NOSUCHFILE, no such file
%LINK-E-FATALERROR, Fatal error message issued
```

"%RMS-F-FNF, file not found" is generated by the VAX-11 RMS system facility.

10.2.2 Run-time Error Messages

When you execute a program, errors can occur as a result of faulty program logic or file I/O problems.

10.2.2.1 **Faulty Program Logic Error Procedures** - If errors occur at run-time, the COBOL-74 run-time system (RTS) displays a message on your terminal. Additionally, the system TRACEBACK facility displays a list of routines that were active when the error occurred.

For example, if you create this program:

```
00001           IDENTIFICATION DIVISION.
00002           PROGRAM-ID. PERF.
00003           ENVIRONMENT DIVISION.
00004           DATA DIVISION.
00005           PROCEDURE DIVISION.
00006           ROUTINE-A.
00007                PERFORM ROUTINE-B.
00008           ROUTINE-B.
00009                PERFORM ROUTINE-C.
00010           ROUTINE-C.
00011                PERFORM ROUTINE-A.
```

it will compile without any detectable errors. However, if you run it, the following will appear on your terminal:

```
$ RUN PERF
%C74-F-RECPERDET, recursive PERFORM detected
%TRACE-F-TRACEBACK, symbolic dump follows

module name      routine          line    relative PC     absolute PC

PERF             PERF              7       0000002C        0000C678
PREF             PERF             11       0000005C        0000C6A8
PERF             PERF              9       00000044        0C00C62F
PERF             PERF              7       0000002C        0C00C678
```

The RTS displays "C74-F-RECPERDET, recursive PERFORM detected" to show that program PERF contains a statement (PERFORM ROUTINE-B) that, if executed, will cause a PERFORM statement to try and perform itself.

If a fatal error occurs, and the program was linked with the /TRACEBACK qualifier (linker default), TRACEBACK will produce a symbolic dump of all call frames that were active when the error occurred. A call frame represents one execution of a subroutine CALL or a PERFORM statement. For each call frame, TRACEBACK displays: (1) the module name (program-id), (2) the routine name (program-id), (3) the source program line number where the error occurred, and (4) program-counter information.

The initial line of the preceding TRACEBACK dump shows that the RTS detected a fatal error on line 7 of program PERF. A loop has been created that will cause the PERFORM on source program line 7 to be executed twice without an intervening EXIT. The remaining lines of the symbolic dump show the sequence in which the PERFORMs were executed, starting with the most recently executed statement.

If program PERF were modified as follows, it would become a callable subprogram:

```
                .
                .
00005           PROCEDURE DIVISION USING.
                .
                .
```

A program could then be written to call PERF:

```
00001           IDENTIFICAION DIVISION.
00002           PROGRAM-ID. DRIVER.
00003           ENVIRONMENT DIVISION.
00004           DATA DIVISION.
00005           PROCEDURE DIVISION.
00006           PARA.
00007                CALL "PERF".
```

If you run DRIVER, the following would appear at your terminal:

```
$ RUN DRIVER
%C74-F-RECPERDET, recursive PERFORM detected
%TRACE-F-TRACEBACK, symbolic dump follows

module name      routine          line     relative PC     absolute PC

PERF             PERF                7       0000002C        0000C678
PERF             PERF               11       0000005C        0000C6A8
PERF             PERF                9       00000044        0000C62F
PERF             PERF                7       0000002C        0000C678
DRIVER           DRIVER              7       0000002F        0000C62F
```

The symbolic dump now contains a fifth line, which shows the calling program DRIVER as the initial entry on the nested PERFORM stack.

10.2.2.2 **File I/O Error Procedures** - If an error occurs during I/O operations, the following procedure is used:

1. If the file status key for the file is present, the RTS sets it to the code for the error condition. Appendix C of the **VAX-11 COBOL-74 Reference** Manual lists file status key values.

2. If an INVALID KEY imperative condition is specified for the I/O operation, the RTS performs the associated imperative statement. The RTS performs no other processing in the file for the current statement. The USE procedure is not performed.

3. If no INVALID KEY imperative condition is specified for the I/O operation and a USE procedure is declared for the file, the RTS performs the USE procedure and returns control to the program. The RTS performs no further processing for the file.

4. If no AT END is specified for a sequential file, and a USE procedure is present, the RTS performs the USE procedure and returns control to the program.

5. If no AT END, and no INVALID KEY, and no USE procedure is declared for the file, an error condition exists; the program terminates with a C74 error status. Both C74 and RMS messages will be displayed as a result.

The following example shows a program that does not contain a USE procedure for a file that is opened for INPUT:

```
00001           IDENTIFICATION DIVISION.
00002           PROGRAM-ID. TESTIO.
00003           ENVIRONMENT DIVISION.
00004           CONFIGURATION SECTION.
00005           SOURCE-COMPUTER. VAX-11.
00006           OBJECT-COMPUTER. VAX-11.
00007           INPUT-OUTPUT SECTION.
00008           FILE-CONTROL.
00009                SELECT NOFILE ASSIGN TO "NOFILE.DAT".
00010           DATA DIVISION.
00011           FILE SECTION.
00012           FD  NOFILE
00013                LABEL RECORDS ARE STANDARD.
00014           01  FILE-REC                PIC X.
00015           PROCEDURE DIVISION.
00016           PARA.
00017                 OPEN INPUT NOFILE.
```

If you execute the program and RMS does not find the file (NOFILE.DAT) in the default directory on the default device, the following messages will appear at your terminal:

```
%C74-F-OPNERRFIL, OPEN error on file: (NOFILE.DAT)
%RMS-E-FNF, file not found
%TRACE-F-TRACEBACK, symbolic stack dump follows

module name      routine name      line    relative PC    absolute PC

TESTIO           TESTIO              17       00000030      0000C630
```

The following example shows a program that does contain a USE procedure for a file that is opened for INPUT:

```
00001           IDENTIFICATION DIVISION.
00002           PROGRAM-ID. TEST2IO.
00003           ENVIRONMENT DIVISION.
00004           CONFIGURATION SECTION.
00005           SOURCE-COMPUTER. VAX-11.
00006           OBJECT-COMPUTER. VAX-11.
00007           INPUT-OUTPUT SECTION.
00008           FILE-CONTROL.
00009                SELECT NOFILE ASSIGN TO "NOFILE.DAT".
00010           DATA DIVISION.
00011           FILE SECTION.
00012           FD  NOFILE
00013                LABEL RECORDS ARE STANDARD.
00014           01  FILE-REC                PIC X.
00015           PROCEDURE DIVISION.
00016           DECLARATIVES.
00017           USE-SECTION SECTION.
00018                USE AFTER STANDARD ERROR PROCEDURE ON INPUT.
00019           USE-IT.
00020                DISPLAY "INVALID FILE OPEN".
00021                STOP RUN.
00022           END DECLARATIVES.
00023           PARA.
00024                OPEN INPUT NOFILE.
```

If you execute this program and the file "NOFILE.DAT" does not exist, the USE procedure will display the following message:

```
"INVALID FILE OPEN".
```

Thus, if you use a USE procedure, the VAX/VMS error facility will not display error messages.

See Appendix E for a full list of RTS error messages.

CHAPTER 11

SORTING IN A COBOL PROGRAM

VAX-11 SORT is a native-mode utility that provides a wide range of sorting capabilities and options; it can be used as an independent utility program or, through COBOL-74, as a set of callable subroutines. VAX-11 SORT is described in detail in the VAX-11 SORT User's Guide.

This chapter introduces the VAX-11 SORT callable subroutines and describes how to use them to sort records in COBOL programs.

11.1 VAX-11 SORT SUBROUTINE PACKAGE

The SORT subroutines are part of the standard VAX/VMS library, and they conform to the VAX/VMS calling standards. Therefore, any native-mode VAX-11 program can call them. In COBOL, you communicate with the subroutines through the CALL statement. The subroutines, because they are in the VMS library, are linked automatically with modules that call them.

VAX-11 SORT provides six subroutines to perform sorting functions:

SOR$PASS_FILES	opens an input file and creates an output file. This routine is used only when files are sorted.
SOR$INIT_SORT	initializes the SORT work areas and work files, using the arguments you pass in the CALL statement. A program calls SOR$INIT_SORT once at the beginning of each sort.
SOR$RELEASE_REC	passes a record to the SORT after your program has processed it. A program calls this subroutine once for each record to be included in the sort.
SOR$SORT_MERGE	performs the sort-merge operation.

SOR$RETURN_REC	returns a record to your program after sorting. A program calls this subroutine once for each record to be returned from the sort.
SOR$END_SORT	performs housekeeping functions at the end of a sort, such as closing files and releasing memory. A program calls SOR$END_SORT once at the end of each sort.

11.2 I/O INTERFACE METHODS

The VAX-11 SORT subroutine package allows you to specify sorts in terms of an entire file or one record at a time; these techniques are called I/O interface methods. This section briefly describes the two I/O interface methods.

11.2.1 File I/O Interface

Using this method, you request VAX-11 SORT to sort all records in a file to create a re-ordered output file. This technique is comparable to the SORT...USING...GIVING syntax of the ANSI-74 COBOL SORT Module.

Call each of the following subroutines once in the order shown:

1. SOR$PASS_FILES
2. SOR$INIT_SORT
3. SOR$SORT_MERGE
4. SOR$END_SORT

The programming example in Section 11.6 uses the file I/O interface method for the second sort operation.

11.2.2 Record I/O Interface

Using this method, your program processes each record before releasing it to the SORT. After all records have been released, they are sorted into the specified order; SORT then returns one record at a time to the program. This technique is functionally identical to the ANSI-74 COBOL SORT with input and output procedures.

Call each of the following routines in the order shown:

1. SOR$INIT_SORT
2. SOR$RELEASE_REC
3. SOR$SORT_MERGE
4. SOR$RETURN_REC
5. SOR$END_SORT

Call SOR$RELEASE_REC and SOR$RETURN_REC once for each record; call the other subroutines only once in each sort.

The programming example in Section 11.6 uses the record I/O interface method for the first sort operation.

11.3 KEY DATA AND RECORD AREAS

For the record I/O interface, the record that you pass to the SOR$RELEASE_REC subroutine consists of the sort keys (key data) followed by the record to be sorted (record area). The key data must contain all the key fields specified in the key buffer, which is described in the next section; furthermore, you should specify the key fields in the same sequence and in the same way (for example, the same size and data type) that they appear in the key buffer. Do not leave space between the key fields.

The record area immediately follows the key data. It defines the record that VAX-11 Sort returns -- the subroutine SOR$RETURN_REC does not return the key data.

You can specify from one to ten keys for each sort. If you need more than ten keys in a single sort, you may be able to combine some key fields to reduce the number of specifications. For example, if the first three keys were all the same type, and if they all were to be sorted in ascending order, you might be able to combine them this way:

Original keys:

```
01  SORT-RECORD.
    03  SORT-KEYS.
        05  SORT-KEY-1    PIC X(10).
        05  SORT-KEY-2    PIC X(5).
        05  SORT-KEY-3    PIC X(20).
        05  SORT-KEY-4.   PIC S9(5) COMP-3.
        .
        .
        .
        05  LAST-KEY      PIC S9(6).
    03  SORT-DATA.
```

Combined keys:

```
01  SORT-RECORD.
    03  SORT-KEYS.
        05  COMBINED-KEY-1.
            07  SORT-KEY-1    PIC X(10).
            07  SORT-KEY-2    PIC X(5).
            07  SORT-KEY-3    PIC X(20).
        05  SORT-KEY-4    PIC S9(5) COMP-3.
        .
        .
        .
        05  LAST-KEY      PIC S9(6).
    03  SORT-DATA.
```

The total size of the key area cannot exceed 255 character positions. However, it is often possible to reduce the size of fields from the record area by using a different data type to specify the key. Compare the storage requirements of the following data descriptions:

```
03  ACCOUNT-NUM PIC 9(11).              Requires 11 characters.
03  ACCOUNT-KEY PIC S9(11) COMP-3.      Requires 6 characters.

03  COST        PIC 9(7)V99.            Requires 9 characters.
03  COST-KEY    PIC 9(7)V99 COMP.       Requires 4 characters.
```

The record I/O interface subroutines allow only three key data types: character, packed-decimal, and word or longword binary. They do not allow other types, such as separate sign, overpunched sign, or quadword binary. However, if you use the record I/O interface, you can define such keys in the key area with one of the allowable key types. In the following examples, the data descriptions on the left are not defined by a valid key type for the record I/O interface subroutines. The data descriptions on the right can be used in the key area for those data items:

Description in record area	Description in key data
PIC S9(5) SIGN LEADING SEPARATE	PIC S9(5) COMP
PIC S9(5) SIGN TRAILING SEPARATE	PIC S9(5) COMP-3
PIC S9(17) COMP	PIC S9(17) COMP-3
PIC S9(5)	PIC S9(5) COMP-3

11.4 KEY BUFFER

The key buffer describes each key to the SOR$INIT_SORT subroutine. Define it as a record (01-level) in the Working-Storage Section.

The first data item in the key buffer specifies the number of individual keys; define it as a one-word COMP data item -- its PICTURE must be in the range 9(1) to 9(4). The following example of the beginning of a key buffer specifies that records will be sorted on three keys:

```
01  KEY-BUFFER.
    03  NUMBER-OF-KEYS    PIC 9(4) COMP VALUE IS 3.
```

Follow the number-of-keys specification with up to ten "blocks" of key definitions. Each block specifies one key field that you defined as key data preceding the record area. The block consists of four one-word COMP data items -- PICTURE 9(1) to 9(4):

key type — Specifies the data type of the key field.

The following data types are valid for the record I/O interface:

1 = character (alphanumeric)
2 = binary (COMPUTATIONAL)
4 = packed-decimal (COMPUTATIONAL-3)

key order — specifies the order for sorting this key field:

0 = ascending
1 = descending

start position — character position in the record (not the key buffer) at which this key field begins. The value of this data item can range from 1 to the maximum record size.

length — specifies the size of the key field in digits, for packed-decimal (COMPUTATIONAL-3) items, or in character positions for all other data items.

For COMPUTATIONAL items, the lengths associated with PICTURE ranges are:

PICTURE	key-length
9(1) to 9(4)	2
9(5) to 9(9)	4
9(10) to 9(18)	cannot be sort key

In the following example, the key buffer specifies three sort keys. The data items in each key definition block are assigned data-names for clarity; however, you can specify them as FILLER if you do not need to refer to them explicitly, since they are passed as a record to SOR$SORT_INIT.

```
WORKING-STORAGE SECTION.
01  KEY-BUFFER.
    03  NUMBER-OF-KEYS        PIC 9(4) COMP VALUE 3.
    03  KEY-1-TYPE            PIC 9(4) COMP VALUE 1.
    03  KEY-1-ORDER           PIC 9(4) COMP VALUE 0.
    03  KEY-1-START           PIC 9(4) COMP VALUE 10.
    03  KEY-1-LENGTH          PIC 9(4) COMP VALUE 25.
    03  KEY-2-TYPE            PIC 9(4) COMP VALUE 4.
    03  KEY-2-ORDER           PIC 9(4) COMP VALUE 1.
    03  KEY-2-START           PIC 9(4) COMP VALUE 1.
    03  KEY-2-LENGTH          PIC 9(4) COMP VALUE 5.
    03  KEY-3-TYPE            PIC 9(4) COMP VALUE 2.
    03  KEY-3-ORDER           PIC 9(4) COMP VALUE 0.
    03  KEY-3-START           PIC 9(4) COMP VALUE 35.
    03  KEY-3-LENGTH          PIC 9(4) COMP VALUE 2.
01  SORT-RECORD.
    03  SORT-KEYS.
        05  KEY-INDUSTRY                PIC X(25).
        05  KEY-NUMBER-OF-EMPLOYEES     PIC S9(5) COMP-3.
        05  KEY-DOLLAR-VOLUME           PIC 9(4) COMP.
    03  SORT-DATA.
        05  NUMBER-OF-EMPLOYEES         PIC S9(5) COMP-3.
        05  FILLER                      PIC X(6).
        05  INDUSTRY                    PIC X(25).
        05  DOLLAR-VOLUME               PIC 9(4) COMP.
        .
        .
        .
```

11.5 SORT SUBROUTINES

Each of the subroutines described in this section performs a separate and necessary function.

The arguments for each subroutine are described as they occur. However, each subroutine returns a longword COMPUTATIONAL result value, which your program can test to detect success and failure conditions. The result status codes are described for each subroutine; however, to make them available to your program, you must include the otherwise optional GIVING phrase in the CALL statements.

For example:

```
CALL "SOR$END_SORT" GIVING SORT-RESULT.
```

causes the result status for the clean-up routine to be available in the COMPUTATIONAL data item, SORT-RESULT, which you have defined in the Working-Storage Section with a PICTURE 9(9).

Sorting is a set of logically ordered procedures, each of which is performed in VAX-11 SORT by a separate subroutine. Therefore, the order in which you call the Sort subroutines is important.

Furthermore, because sorting is a set of procedures, you must complete one sort before beginning another. You can have as many separate sorts as you need in a single COBOL program; however, if you do not complete a sequence of sort subroutine calls before starting another, an error results.

The following Sort subroutines are discussed in the order that they must be called.

11.5.1 SOR$PASS_FILES

For the file I/O interface, this subroutine passes the names of the input and output files to VAX-11 Sort.

The general form of the CALL is:

```
CALL "SOR$PASS_FILES"
    USING BY DESCRIPTOR
        <input file>
        <output file>
    [GIVING <result status>]
```

Arguments

input file	is the data-name of a data item that contains the file specification (or logical name) of the input file.
output file	is the data-name of a data item that contains the file specification (or logical name) of the output file.

Result Status Values

Symbolic	Hex Value	Decimal Value	Meaning
NORMAL	1	1	Success.
SORT_ON	1C802C	1867820	A sort is already in progress or this call is in the wrong sequence.
VAR_FIX	1C8064	1867876	You cannot change variable-length records to fixed-length records.
INCONSIS	1C805C	1867868	Inconsistent data for file.
OPENIN	1C109C	1839260	Cannot open input file.
OPENOUT	1C10A4	1839268	Cannot open output file.
All RMS error codes.			

11.5.2 SOR$INIT_SORT

This subroutine begins a sort. It initializes the Sort's work files and areas, and it interprets the parameters (arguments) that are passed by the program.

The general form of the CALL is:

```
CALL "SOR$INIT_SORT"
    USING
        <key buffer>
        <LRL>
        [ <file size> ]
        [ <work files> ]
    [GIVING <result status>]
```

Arguments

key buffer — is the data-name of the key buffer, which you have defined in the Working-Storage Section. The key buffer is discussed in Section 11.3.2.

LRL — is the longest record length - a one-word COMP data item, which you have defined in the Working-Storage Section, that specifies the longest record length (in character positions). Record length does not include the key area.

File size — is the data-name of a one-word COMP data item that specifies the size, in blocks, of the input file. This argument is not required, but it can increase Sort efficiency.

work files — is the data-name of a one-word COMP data item that specifies the number of work files the Sort should use. Valid values are 0 and 2 through 10; the default is 2. The Sort expects this argument as a one-byte binary item; however, define it in COBOL as a one-word COMPUTATIONAL item - PICTURE 9(1) to 9(4).

Result Status Values

Symbolic	Hex Value	Decimal Value	Meaning
NORMAL	1	1	Success.
SORT_ON	1C802C	1867820	A sort is already in progress or this call is in the wrong sequence.
MISS_KEY	1C8004	1867780	No key definition specified.
BAD_TYPE	1C806C	1867884	An invalid sort process was specified.
BAD_LRL	1C8084	1867908	An invalid LRL was specified.
LRL_MISS	1C8074	1867892	No LRL was specified.
BAD_FILE	1C808C	1867916	Invalid file size.
WORK_DEV	1C80CC	1867788	Work file device not random access or not local node.
VM_FAIL	1C801C	1867804	SORT failed to get needed virtual memory.
WS_FAIL	1C8024	1867812	SORT failed to get needed working-set size.
NUM_KEY	1C803C	1867836	Invalid number of keys specified. Must be 1-10.
KEY_LEN	1C80AC	1867948	Invalid key length specified.
All RMS error codes.			

11.5.3 SOR$RELEASE_REC

This subroutine passes, or releases, a record to VAX-11 Sort for the record I/O interface. Before calling SOR$RELEASE_REC, your program must construct the sort keys in the key data area. Usually, constructing the keys involves nothing more than moving their values from the fields in the record area to the fields in the key area. An exception might be when you construct keys by combining data items in the record area, or if you compute key values in some other way.

Call this subroutine once for each record you want to be included in the sort. This call is comparable to the ANSI COBOL RELEASE statement in an Input Procedure.

The general form of the call is:

```
CALL "SOR$RELEASE_REC"
    USING
        BY DESCRIPTOR <key data>
    [GIVING <result status>]
```

Argument

key data — is the record-name of the key data area. The key data record-name includes both the key data and the record area; therefore, this argument gives the subroutine all the information it needs to access both the sort keys and the data.

Result Status Values

Symbolic	Hex Value	Decimal Value	Meaning
NORMAL	1	1	Success.
SORT_ON	1C802C	1867820	A sort is already in progess or this call is in the wrong sequence.
BAD_LRL	1C8084	1867908	Record length is longer than the LRL that was specified to SOR$INIT_SORT.
BAD_ADR	1C8094	1867924	Invalid key area address.
KEY_LEN	1C80AC	1867948	Invalid key length specified.
EXTEND	1C80A4	1867940	Failed to extend work file.
MAP	1C809C	1867932	Internal Sort map error.
NO_WRK	1C8014	1867796	Cannot sort data in memory. Need work files.

11.5.4 SOR$SORT_MERGE

This subroutine performs the final phases of the sort-merge process. For the record I/O interface, call it once, after the last record has been released to the Sort, and before attempting to return the first of the sorted records. For the file I/O interface, call SOR$SORT_MERGE once after calling SOR$INIT_SORT.

The general form of the call is:

```
CALL "SOR$SORT_MERGE"
    [GIVING <result status>]
```

Arguments

None.

Result Status Values

Symbolic	Hex Value	Decimal Value	Meaning
NORMAL	1	1	Success.
SORT_ON	1C802C	1867820	A sort is already in progress, or this call is in the wrong sequence.
EXTEND	1C80A4	1867940	Failed to extend work file.
NO_WRK	1C8014	1867796	Cannot sort data in memory. Need work files.
MAP	1C809C	1867932	Internal Sort map error.
READERR	1C10B4	1839284	Cannot read an input file record.
WRITRERR	1C10D4	1839316	Cannot write an output file record.
BADFIELD	1C101C	1839132	Bad data in key field.

11.5.5 SOR$RETURN_REC

This subroutine returns one record to your program from the Sort. It places the record in the record area data item; it also returns the record length.

You cannot call this subroutine before calling the sort-merge subroutine. Call SOR$RETURN_REC once for each record to be returned from the Sort.

The general form of the call is:

```
CALL "SOR$RETURN_REC"
    USING BY DESCRIPTOR <record area>
          BY REFERENCE <record length>
    [GIVING <result status>]
```

Arguments

record area	is the data-name of the area into which the returned record should be placed. Usually, it is the same area from which it was released; however, you can specify another data-name.
record length	is the data-name of a one-word COMPUTATIONAL data item into which the subroutine will place the actual size of the returned record.

Result Status Values

Symbolic	Hex Value	Decimal Value	Meaning
NORMAL	1	1	Success. A record has been returned.
ENDOFFILE	870	2160	No more records to return.
MAP	1C809C	1867932	Internal Sort map error.
EXTEND	1C8084	1867940	Failed to extend work file.

11.5.6 SOR$END_SORT

This subroutine deletes the Sort's work files and releases its work areas. You must call SOR$END_SORT before beginning another sort; however, it is good programming practice to call this subroutine at the end of any sort to release work file space and memory.

The general form of the call is:

```
CALL "SOR$END_SORT"
    [GIVING <result status>]
```

Arguments

None.

Result Status Values

Symbolic	Hex Value	Decimal Value	Meaning
NORMAL	1	1	Success.
CLEAN_UP	1C80B4	1867956	Failed to delete work files and reinitialize work areas.

11.6 PROGRAMMING EXAMPLE

The program in this section reads a sequential mailing list file, attempts to detect duplicates, and writes a new file. It uses the record I/O interface Sort technique, after constructing an artificial identification key, to return identically-keyed records together, so they can be compared. After it writes the new file, the program uses the file I/O interface to sort the file into its original order.

One of the sort keys (subscription start date) is specified as descending order, because the designer assumes that the earliest record is probably the most accurate.

The comparable ANSI-74 COBOL SORT module statements are included as comments for comparison.

```
      IDENTIFICATION DIVISION.
      PROGRAM-ID.
          MLIST.
      DATE-WRITTEN.
      DATE-COMPILED.
      ENVIRONMENT DIVISION.
      CONFIGURATION SECTION.
      SOURCE-COMPUTER.     VAX-11.
      OBJECT-COMPUTER.     VAX-11.
      INPUT-OUTPUT SECTION.
      FILE-CONTROL.
          SELECT MAILING-FILE
            ASSIGN TO "MAILLIST".
          SELECT NEW-FILE
            ASSIGN TO "NEWLIST".
     *    SELECT SORT-FILE
     *      ASSIGN TO "SORTF".
       DATA DIVISION.
       FILE SECTION.
     *SD  SORT-FILE
     *    LABEL RECORDS ARE OMITTED.
     *01  SORT-REC.
     *    03  S-ZIP                PIC 9(5).
     *    03  S-LAST-NAME.
     *        05  S-NAME-1         PIC X.
     *        05  S-NAME-2         PIC X.
     *        05  FILLER           PIC X.
     *        05  S-NAME-4         PIC X.
     *        05  FILLER           PIC X(12).
     *    03  S-FIRST-NAME         PIC X(12).
     *    03  S-STREET.
     *        05  S-STREET-KEY     PIC X(4).
     *        05  FILLER           PIC X(16).
     *    03  S-CITY.
     *        05  S-CITY-KEY       PIC X(4).
     *        05  FILLER           PIC X(16).
     *    03  FILLER               PIC XX.
     *    03  S-START              PIC 9(6).
       FD  MAILING-FILE
          LABEL RECORDS ARE STANDARD.
       01  MAILING-REC.
          03  MAILING-KEY.
              05  ZIP-CODE         PIC 9(5).
              05  LAST-NAME.
                  07  LAST-NAME-CHAR
                                   OCCURS 16 PIC X(1).
              05  FIRST-NAME       PIC X(16).
              05  STREET           PIC X(20).
              05  CITY             PIC X(20).
          03  STATE                PIC X(2).
          03  SUBSCRIP-START       PIC 9(6).
```

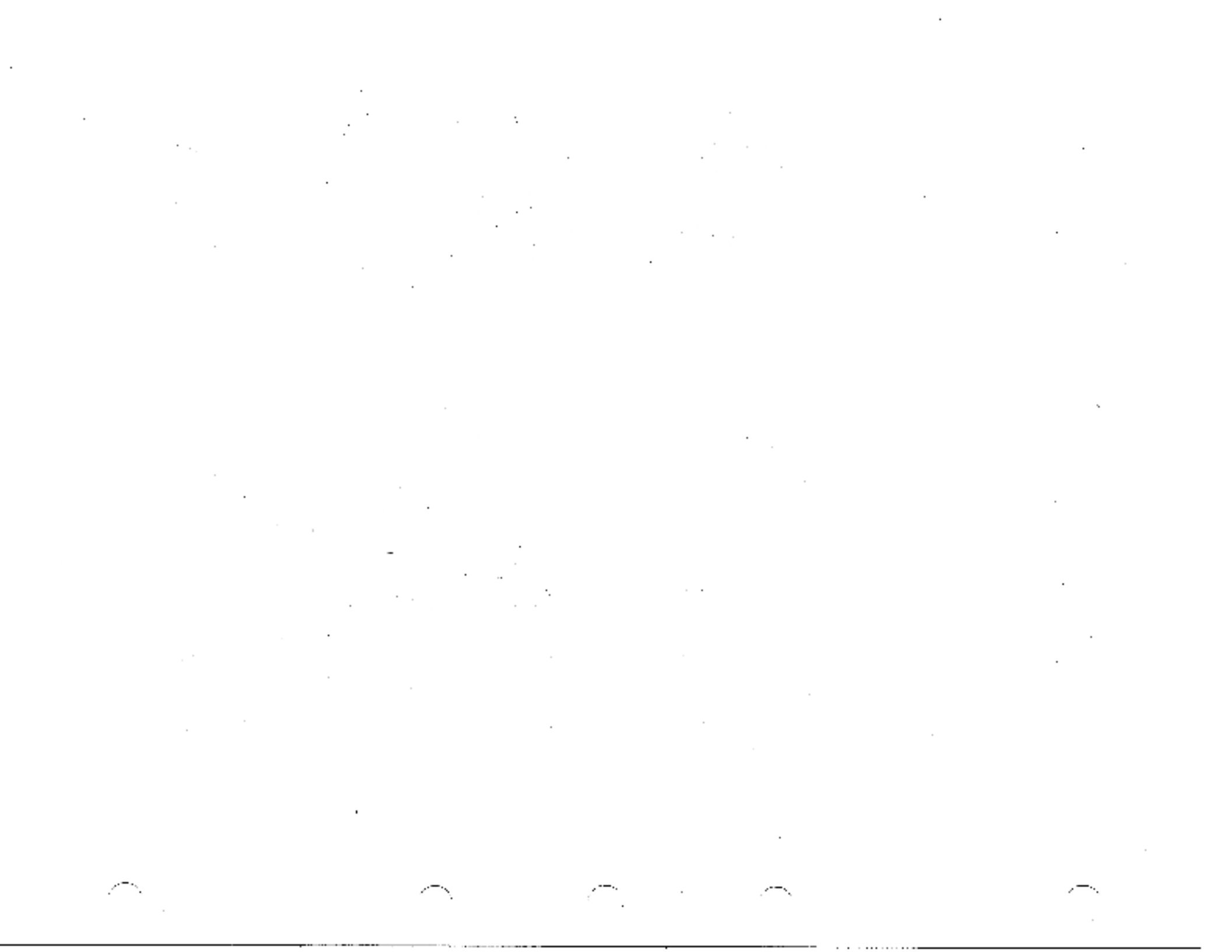
```
FD  NEW-FILE
    LABEL RECORDS ARE STANDARD.
01  NEW-REC                PIC X(85).

WORKING-STORAGE SECTION.

01  MAILING-FILE-ID        PIC X(8) VALUE "MAILLIST".
01  NEW-FILE-ID            PIC X(8) VALUE "NEWLIST".

01  FIRST-IN               PIC X(85) VALUE SPACES.
01  FIRST-KEY.
    03  FIRST-COMPARE      PIC X(14) VALUE SPACES.
    03  FILLER             PIC X(6) VALUE SPACES.

01  SORT-RECORD.
    03  SORT-KEYS.
        05  KEY-ZIP            PIC S9(5) COMP-3.
        05  NAME-ADDRESS-GROUP.
            07  KEY-LAST.
                09  KEY-LAST-CHAR
                                   OCCURS 3 PIC X(1).
            07  KEY-STREET     PIC X(4).
            07  KEY-CITY       PIC X(4).
        05  KEY-START          PIC 9(6).
    03  SORT-DATA.
        05  LAST-IN            PIC X(85).
        05  LAST-KEY.
            07  LAST-COMPARE   PIC X(14).
            07  FILLER         PIC X(6).

01  KEY-BUFFER.
    03  NUMBER-OF-KEYS     PIC 9(4) COMP VALUE 3.

    03  FILLER             PIC 9(4) COMP VALUE 4.
    03  FILLER             PIC 9(4) COMP VALUE 0.
    03  FILLER             PIC 9(4) COMP VALUE 75.
    03  FILLER             PIC 9(4) COMP VALUE 5.

    03  FILLER             PIC 9(4) COMP VALUE 1.
    03  FILLER             PIC 9(4) COMP VALUE 0.
    03  FILLER             PIC 9(4) COMP VALUE 1.
    03  FILLER             PIC 9(4) COMP VALUE 11.

    03  FILLER             PIC 9(4) COMP VALUE 1.
    03  FILLER             PIC 9(4) COMP VALUE 1.
    03  FILLER             PIC 9(4) COMP VALUE 80.
    03  FILLER             PIC 9(4) COMP VALUE 6.

01  ORIGINAL-KEY-BUFFER.
    03  FILLER             PIC 9(4) COMP VALUE 1.

    03  FILLER             PIC 9(4) COMP VALUE 1.
    03  FILLER             PIC 9(4) COMP VALUE 0.
    03  FILLER             PIC 9(4) COMP VALUE 1.
    03  FILLER             PIC 9(4) COMP VALUE 77.
```

```
01  RESULT-STAT          PIC 9(8) COMP.
    88  SUCCESSFUL  VALUE 1.
    88  END-SORT    VALUE 2160.
01  LRL                  PIC 9(4) COMP VALUE 105.
01  REC-LENGTH          PIC 9(4) COMP.

01  FILE-STAT           PIC 9(1) VALUE 0.
    88  ENDFILE   VALUE 1.
01  DUPLICATES-DELETED  PIC 9(6) COMP VALUE 0.
01  DISPLAY-DUPLICATES  PIC ZZZ,ZZ9.
01  DISPLAY-RESULT      PIC 9(8).

PROCEDURE DIVISION.
MAINLINE SECTION.
SBEGIN.
    OPEN INPUT MAILING-FILE.

*   SORT SORT-FILE
*       ASCENDING  S-ZIP
*                  S-NAME-1
*                  S-NAME-2
*                  S-NAME-4
*                  S-STREET-KEY
*                  S-CITY-KEY
*       DESCENDING S-START
*           INPUT PROCEDURE GET-AND-RELEASE
*           OUTPUT PROCEDURE RETURN-AND-WRITE.

    CALL "SOR$INIT_SORT"
      USING
        KEY-BUFFER
        LRL
      GIVING RESULT-STAT.

    IF NOT SUCCESSFUL
      DISPLAY "INIT-SORT ERROR"
      GO TO ABORT-RUN.

    PERFORM GET-AND-RELEASE
      UNTIL ENDFILE.

    CLOSE MAILING-FILE.

    CALL "SOR$SORT_MERGE"
      GIVING RESULT-STAT.

    IF NOT SUCCESSFUL
      DISPLAY "SORT-MERGE ERROR"
      GO TO ABORT-RUN.

    OPEN OUTPUT NEW-FILE.

    PERFORM RETURN-AND-WRITE
      UNTIL END-SORT.

    CALL "SOR$END_SORT"
      GIVING RESULT-STAT.
```

```
      IF NOT SUCCESSFUL
        DISPLAY "END-SORT ERROR"
        GO TO ABORT-RUN.

      MOVE DUPLICATES-DELETED TO DISPLAY-DUPLICATES.
      DISPLAY DISPLAY-DUPLICATES " Duplicates deleted".
      CLOSE NEW-FILE.

*     SORT SORT-FILE
*         ASCENDING S-ZIP
*                   S-LAST-NAME
*                   S-FIRST-NAME
*                   S-STREET
*                   S-CITY
*         USING NEW-FILE
*         GIVING MAILING-FILE.

      CALL "SOR$PASS_FILES"
        USING
          BY DESCRIPTOR
              NEW-FILE-ID
              MAILING-FILE-ID
        GIVING RESULT-STAT.

      IF NOT SUCCESSFUL
        DISPLAY "PASS-FILES ERROR"
        GO TO ABORT-RUN.

      CALL "SOR$INIT_SORT"
        USING
          ORIGINAL-KEY-BUFFER
          LRL
        GIVING RESULT-STAT.

      IF NOT SUCCESSFUL
        DISPLAY "INIT-SORT ERROR"
        GO TO ABORT-RUN.

      CALL "SOR$SORT_MERGE"
        GIVING RESULT-STAT.

      IF NOT SUCCESSFUL
        DISPLAY "SORT-MERGE ERROR"
        GO TO ABORT-RUN.

      CALL "SOR$END_SORT"
        GIVING RESULT-STAT.

      IF NOT SUCCESSFUL
        DISPLAY "END-SORT ERROR".

      STOP RUN.
```

```
 GET-AND-RELEASE SECTION.
 SBEGIN.
     MOVE SPACES TO MAILING-REC.
     READ MAILING-FILE
       AT END
         MOVE 1 TO FILE-STAT
         GO TO SEXIT.

     MOVE MAILING-REC TO SORT-DATA.

     MOVE LAST-NAME-CHAR (1) TO KEY-LAST-CHAR (1).
     MOVE LAST-NAME-CHAR (2) TO KEY-LAST-CHAR (2).
     MOVE LAST-NAME-CHAR (4) TO KEY-LAST-CHAR (3).
     MOVE STREET TO KEY-STREET.
     MOVE CITY TO KEY-CITY.
     MOVE ZIP-CODE TO KEY-ZIP.
     MOVE SUBSCRIP-START TO KEY-START.
     MOVE NAME-ADDRESS-GROUP TO LAST-KEY.

*    RELEASE SORT-REC FROM MAILING-REC.

     CALL "SOR$RELEASE_REC"
       USING
         BY DESCRIPTOR SORT-RECORD
       GIVING RESULT-STAT.

     IF NOT SUCCESSFUL
       DISPLAY "RELEASE-REC ERROR"
       GO TO ABORT-RUN.

 SEXIT.
     EXIT.

 RETURN-AND-WRITE SECTION.
 SBEGIN.

*    RETURN SORT-FILE INTO SORT-DATA
*      AT END
*        MOVE SPACES TO LAST-KEY
*        GO TO COMPARE-KEYS.
*    MOVE (data to keys for comparison).


     CALL "SOR$RETURN_REC"
       USING
         BY DESCRIPTOR SORT-DATA
         BY REFERENCE REC-LENGTH
       GIVING RESULT-STAT.

     IF END-SORT
       MOVE SPACES TO LAST-KEY
       GO TO COMPARE-KEYS.

     IF NOT SUCCESSFUL
       DISPLAY "RETURN-REC ERROR"
       GO TO ABORT-RUN.
```

```
COMPARE-KEYS.
    IF LAST-COMPARE NOT = FIRST-COMPARE
      AND FIRST-COMPARE NOT = SPACES
        WRITE NEW-REC FROM FIRST-IN
    ELSE
      IF FIRST-KEY NOT = SPACES
        ADD 1 TO DUPLICATES-DELETED.

    MOVE LAST-KEY TO FIRST-KEY.
    MOVE LAST-IN TO FIRST-IN.

$EXIT.
    EXIT.

ABORT-RUN SECTION.
$BEGIN.
    MOVE RESULT-STAT TO DISPLAY-RESULT.
    DISPLAY DISPLAY-RESULT.
    STOP RUN.
```



CHAPTER 12

USING THE LIBRARY FACILITY

The VAX-11 COBOL-74 library facility allows you to copy COBOL source language text from a library file into your COBOL program during compilation. One COPY statement can include large amounts of library source text in a program, eliminating a great deal of repetitious coding and the errors that often go along with it. The compiler treats the copied text as if it were a part of the source program; however, the copied material does not change the source program file in any way.

The COBOL library facility provides two important benefits:

1. Standardization of File and Coding Conventions

   A data file is usually processed by more than one program. Each of those programs must describe the characteristics of the file, such as file-name, blocking factor and record descriptions. The programs are often written by one programmer, then maintained and updated by another. Because it is often difficult for a programmer to understand a program written by someone else, many organizations design and code standardized file descriptions, then keep them in COBOL libraries; programmers then COPY the file descriptions into their programs, frequently without having to understand (or even know) their details.

   This technique also applies to Procedure Division code that is used in many different programs. For example, a library could contain a standardized routine to convert calendar dates to Julian dates, or to format standard report headings.

2. Saving Time and Reducing Errors

   Defining and coding file and record descriptions are both time-consuming and error-prone activities. When the descriptions already exist in COBOL libraries, you can easily COPY them into a source program; you save time because you don't have to code them again, and you avoid potential errors in re-entering complex code.

Changing the format of a file is another common time-consuming chore. When a file format changes, you usually must change and recompile all programs that use the file. If the file description is in a COBOL library, only the library must be changed; individual programs often then need only recompilation, since the library coding changes are included by the COPY.

Putting commonly used Procedure Division code in libraries yields the same benefits.

12.1 Creating a COBOL Library File

Each line of a COBOL library file must form syntactically correct COBOL text when it is merged into the source program. It can meet this condition by being itself syntactically correct or by becoming correct when it is merged with the source program.

Library text must conform to the rules for the COBOL source reference format; for example, library text that will appear in Area A of the source program must be in Area A in the library file. You can write library text using either the conventional format or terminal format; however, the library text format must be the same as the source program into which it is merged.

12.2 The COPY Statement

COPY is a compiler-directing statement that merges a COBOL library file into a COBOL source program. The simplest form of the statement is:

```
COPY text-name.
```

Text-name must be either an alphanumeric literal or a file name. Remember that the COPY statement must end with a terminator period regardless of where it appears in the source program.

If you specify a literal, the compiler uses its value as a file specification; therefore, you can include or omit all components of the file specification that are allowed in the VAX/VMS command language, such as device, directory, file type, and version number. The only required component is the file name itself.

For example:

```
COPY "[ACCTLIB]ACCFIL.XYZ;3".
```

causes the compiler to access version number 3 of the file ACCFIL.XYZ in directory [ACCTLIB] on the default device.

If you use a file name in the COPY statement, the compiler uses .LIB as the default file type.

For example:

```
COPY ACCOUNT.
```

causes the compiler to access the latest version of the file ACCOUNT.LIB on the default device and directory.

Only four conditions require the use of the alphanumeric literal to indicate the full file specification for the copy statement:

1. When the file type is other than .LIB.
2. When the library file is not on the default device.
3. When the library file is not in the default directory.
4. When the default directory contains more than one version of the library file and you want to copy a version other than the latest.

Figure 12-1 demonstrates the use of the COPY statement to include Procedure Division code. Note that the format of the library text is maintained when it is included in the source program.

COBOL Source Program

```
...

PROCEDURE DIVISION.
START-PROC SECTION.
BEGIN-PROC.
    ACCEPT TO-DATE
      FROM DATE.
OPEN-FILES. COPY OPENF.
    OPEN I-O WORK-FILE.
INPUT-LOOP.
    READ COST-FILE ...
```

Library File (OPENF.LIB)

```
*
    OPEN INPUT COST-FILE.
    OPEN I-O ORDERS.
GET-VERSION.
    DISPLAY "VERSION?".
    ACCEPT VER-NUM.
    IF VER-NUM NOT NUMERIC
      GO TO GET-VERSION.
*
```

Resulting Source Program

```
...

PROCEDURE DIVISION.
START-PROC SECTION.
BEGIN-PROC.
    ACCEPT TO-DATE
      FROM DATE.
OPEN-FILES. COPY OPENF.
*
    OPEN INPUT COST-FILE.
    OPEN I-O ORDERS.
GET-VERSION.
    DISPLAY "VERSION?".
    ACCEPT VER-NUM.
    IF VER-NUM NOT NUMERIC
      GO TO GET-VERSION.
*
    OPEN I-O WORK-FILE.
INPUT-LOOP.
    READ COST-FILE ...
```

Figure 12-1 Merging Library Text

The COPY statement can appear anywhere that a COBOL word is allowed in a source program; therefore, you can use it in many ways to solve different problems. For example, if a library file called MTG contains the single entry MORTGAGE-PAYMENT-AMOUNT, it could be copied in the Data Division:

```
Source Statement:    03 COPY MTG. PIC 999V99.

Resulting
Source Statement:    03 MORTGAGE-PAYMENT-AMOUNT PIC 999V99.
```

or in the Procedure Division:

```
Source Statement:    MULTIPLY COPY MTG. BY 12
                       GIVING ANNUAL-PAYMENT.

Resulting
Source Statement:    MULTIPLY MORTGAGE-PAYMENT-AMOUNT BY 12
                       GIVING ANNUAL-PAYMENT.
```

The periods following the COPY statements in these examples do not become part of the source text. If the library text requires punctuation, it must be included in the library file.

NOTE

The two preceding examples are not recommended uses of the COPY statement. They are included only to illustrate the mechanics of the COBOL library facility.

12.3 The COPY REPLACING Statement

It is sometimes necessary to tailor library file text for use in a particular program. For example, if a record description in a library file has level-numbers incremented by 1 (01, 02, 03, ...) and you want them to be incremented by four (01, 05, 09, ...), you can change the level-numbers as the library text is merged into the source program. During the copying process, the COPY statement can replace all occurrences of a literal or word with an alternate literal or word. For example:

```
COPY ACCTREC REPLACING 02 BY 05,
  03 BY 09, 04 BY 13.
```

This sample statement causes the compiler to scan the file ACCTREC searching for the character-string 02. Wherever it finds a 02, the compiler substitutes 05. A match occurs only if the compiler finds a 02; no match occurs for a 0 or a 2 alone. The compiler follows the same procedure for occurrences of 03 and 04.

The following examples COPY the library file named NEWSBOY, which contains this text:

```
01  A.
    02  B PIC 99.
    02  C PIC 99 VALUE 2.
    02  D PIC X(5) VALUE "ABCDE".
    02  E PIC 99V99 VALUE 3.75.
    02  F PIC 99 VALUE 02.
```

Example 1

Statement:

```
COPY NEWSBOY REPLACING B BY X.
```

Result:

```
01  A.
    02  X PIC 99.
    02  C PIC 99 VALUE 2.
    02  D PIC X(5) VALUE "ABCDE".
    02  E PIC 99V99 VALUE 3.75.
    02  F PIC 99 VALUE 02.
```

Example 2

Statement:

```
COPY NEWSBOY REPLACING 2 BY 6.
```

Result:

```
01  A.
    02  B PIC 99.
    02  C PIC 99 VALUE 6.
    02  D PIC X(5) VALUE "ABCDE".
    02  E PIC 99V99 VALUE 3.75.
    02  F PIC 99 VALUE 02.
```

Example 3

Statement:

```
COPY NEWSBOY REPLACING 02 BY 63.
```

Result:

```
01  A.
    63  B PIC 99.
    63  C PIC 99 VALUE 2.
    63  D PIC X(5) VALUE "ABCDE".
    63  E PIC 99V99 VALUE 3.75.
    63  F PIC 99 VALUE 63.
```

In the last example, level-number 02 was changed to level-number 63, which is not legal under COBOL rules; therefore, although both the COPY statement and the library text are syntactically correct, the merged text is incorrect and would generate syntax errors.

The REPLACING character-string can be a literal or a word; it must compare equally, character for character, with the entire character-string in the library text. Table 12-1 illustrates the results of some character-string comparisons.

REPLACING Literal or Word	Library Text	Match?
"ABC"	"ABCD"	No
HRLY-RATE	HRLY-RATE	Yes
1	1	Yes
"2"	2	No
" 15"	"15"	No
"012"	"12"	No
012	12	No
SUBTRACT	SUBTRACT	Yes
"012"	"012"	Yes
ACCT	ACCT1	No

Table 12-1 COPY REPLACING Matches

12.4 The Source Listing

Depending on how you write the COPY statement, library text can appear either before or after the COPY statement. The compiler normally prints a line of source text when it scans to the end of the line; however, when the compiler recognizes a completed COPY statement before the end of the line, it locates the library file, then:

1. Prints the library text.
2. Scans the rest of the source program line.
3. Prints the entire source line.

Thus, if the source line contains a COPY statement followed by other text (including spaces), the compiler prints the library text before the source line containing the COPY statement; this results in a somewhat confusing listing. You can cause the compiler to produce a more readable listing by making sure that you write each COPY statement as the last entry on a source program line.

12.5 Common Errors in Using the Library Facility

Some of the more common errors to avoid when using the library facility are:

- Failing to follow the rules for the COBOL reference format when creating the library file.
- Merging a library file in one format (conventional or terminal) with a source program written in the other.
- Forgetting to end the COPY statement with a terminator period.
- Inadvertently defining data-names in the source program when they are also defined in the library file, thus causing duplicate names.
- Writing library file text that becomes syntactically incorrect when it is merged with the source program.
- Merging the wrong library file, either because multiple versions exist, or because of misspellings.
- Writing source text following the COPY statement on the same line, thus causing confusion in the source program listing.
- Forgetting that numeric literals (such as 02, 77, ...) used in the REPLACING option replace level-numbers, picture descriptions, and paragraph or section names, when they find matches in the library file.
- Forgetting that a period must appear in the library file if it is to appear in the source program; the terminator period that ends the COPY statement is replaced by library text.

CHAPTER 13

OPTIMIZATION

Optimization is the process of designing or altering a program to minimize space allocation or execution time, or to achieve an effective trade-off between the two.

This chapter provides guidelines for optimizing performance of COBOL programs. It emphasizes techniques, controllable at the COBOL source level, for optimizing file design, program organization, and computation. Many COBOL programs make heavy use of file I/O. Consequently, your methods of designing, populating, and handling files can either enhance or undermine system performance.

When optimizing COBOL programs, aim to minimize I/O activity. You can accomplish this by the way you design files and structure your program. Your answers to the following questions should influence your choice of file organization, record type, buffer size and number, and your program organization:

1. What kinds of I/O operations are necessary to process the data?

2. How can you best place I/O operations in the program?

3. How should you structure the file? Are multiple access keys necessary or desirable?

4. For each file, are frequent record updates and insertions likely, or will file contents remain relatively (or absolutely) stable?

You can also influence computational performance, especially by formatting data to avoid data conversions and utilize fast, specialized computational routines of the compiler.

The following sections describe each of these optimization techniques.

NOTE

For more information on optimization techniques you can use through Record Management Services (RMS) facilities, refer to appropriate RMS documentation.

13.1 OPTIMIZING FILE DESIGN

This section describes the effect of file design on performance. The following suggestions apply to any type of file organization.

1. Preallocate the entire file, contiguously if possible, using the /CO:n or /AL:n file switch (see Table 6-2) or the RMS DEFINE utility.

2. Select a suitable default extend quantity when you create the file, using the /EX:n file switch or the RMS DEFINE utility. (Refer to RMS documentation for a description of default extend quantities and the RMS DEFINE utility.)

3. Know the relationships between record size and file storage, and try to define a record size suited for efficient storage and retrieval.

4. Use the SAME RECORD AREA clause to save compute time and conserve address space. If records are being copied from one file to another, and both files share the same record area, no MOVE statement is needed to move record images between two record areas. The disadvantage is that records from both files cannot be available simultaneously unless one is moved to a work area. (Be careful not to confuse the SAME RECORD AREA and SAME AREA clauses; they appear similar, but have different effects.)

13.1.1 Sequential Files

Sequential files have the simplest structure and the fewest options for definition, population, and handling. You can reduce the number of disk accesses by keeping record length to a minimum.

With a sequential disk file, you can use the multi-block read and write facility to create a larger buffer area. To use this facility, specify the BLOCK CONTAINS n CHARACTERS clause in combination with ORGANIZATION IS SEQUENTIAL. If you omit the BLOCK CONTAINS n CHARACTERS clause, the RMS default applies.

13.1.2 Relative Files

For relative files:

1. Select a record format and size that minimizes the empty space remaining in each record position and each bucket.

2. If you create the file by using the RMS DEFINE utility, select a realistic maximum record number. An attempt to insert a record with a number higher than the maximum will fail. Before inserting such a record, you must redefine and repopulate the file.

3. Be aware that, before writing a record into a relative file, RMS must have formatted all buckets up to and including the bucket into which the record insertion will occur. Thus, write operations have variable response times, depending on whether preliminary formatting is required, and how much. You might consider writing the highest-numbered record first to force formatting of the entire file only once.

13.1.3 Indexed Files

Indexed files have the greatest potential for inefficient usage. Therefore, carefully consider how well the design and use of the files map into the application. To do this, you must first understand how indexed files are organized and processed.

As the name suggests, an indexed file contains, besides data records, pointer information to facilitate access to the records.

All data records and record pointers are maintained in storage units called buckets. The bucket is the basic retrievable element of an indexed file. It consists of an integral number of contiguous 512-byte physical blocks, and the number of physical blocks is known as the bucket size.

Every indexed file must have a primary key: a field in the record description that contains a unique value for each individual record. When RMS writes records into the indexed file, it arranges them in collated sequence, according to increasing primary key value, in a series of chained buckets. Thus, you can access the records sequentially, if you wish, by specifying ACCESS SEQUENTIAL.

As RMS writes the records, it constructs and maintains a tree-like structure of key-value and location pointers. (See Figure 13-1.) Each element of the index structure is a bucket, and the buckets are structured into a hierarchy of levels. The highest level of the index consists of a single bucket, called the root bucket. The root bucket contains location pointers to buckets at the next lower level. Thus, RMS scans one bucket at each level of the index for a pointer to a bucket at the next level, until it reaches the bottom level of the index; the bottom level is called the data level. In a primary key index, this level contains the actual data records of the indexed file. The buckets in each level above the data level are called index buckets.

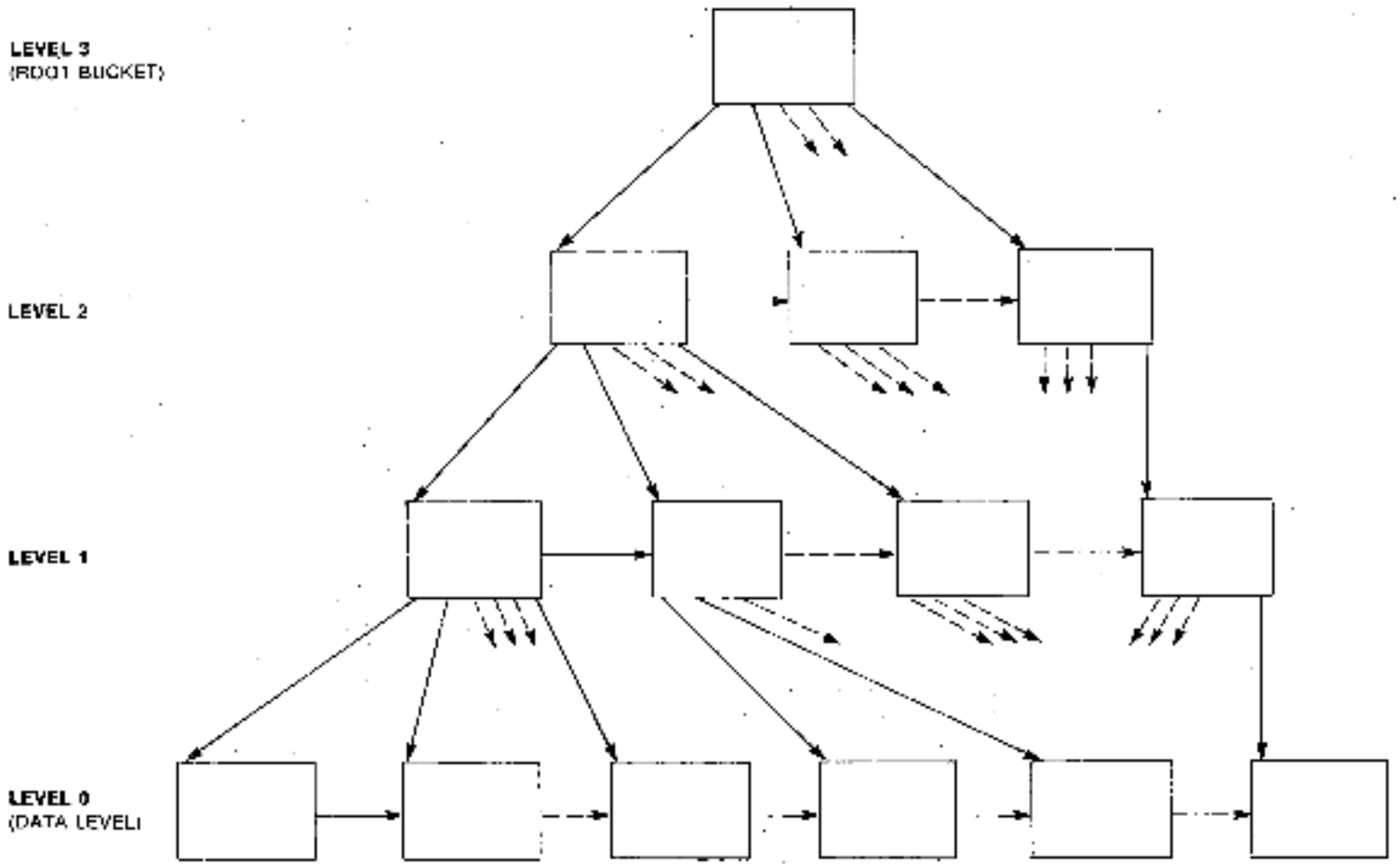


Figure 13-1 Three-Level Primary Key Index

RMS also constructs an index for each alternate key that you define for the file. Like the primary index, alternate key indexes are contained in the file. However, alternate key indexes do not contain actual data records at the data level; instead, they contain pointers to data records in the data level of the primary index.

Successive levels of an index are numbered. The data level of the index is level zero, and the number of levels above level zero is the index depth. Thus, the level number of the root bucket equals the depth of the index.

Each random access request begins by comparing a specific key value against the entries in the root bucket, seeking the first entry in the root bucket whose key value equals or exceeds the value of the access request key. (This search is always successful, because the root bucket's highest key value is the highest possible value that the key field can contain.) Having located the proper key value, RMS uses the bucket pointer associated with that value to bring the target bucket on the next lower level into memory. This process is repeated for each level of the index. RMS thus searches one bucket at each level of the index until it reaches a target bucket at the data level. At this point, the desired data record location is determined; an existing data record can be retrieved or deleted, or a new record written. Duplicate primary key values are not allowed; if a record insertion would cause a duplicate primary key value, the attempted write causes an exception condition.

There may be insufficient room in a data level bucket to accommodate a new record. When this occurs, RMS inserts a new bucket in the chain, moving enough records from the old bucket to preserve the key value sequence, while making room to write the new record. This action is known as a bucket split.

In summary, each index of an indexed file provides the mechanism for random access to records. Sequential access to records is also possible, because the records of the primary index, or pointers of an alternate index, are collated in ascending key value order.

13.1.3.1 **General Rules for Indexed Files** - You can apply the following general rules for indexed files at the COBOL source code level.

1. While alternate keys are often useful, the more keys you define for an indexed file, the longer each WRITE , REWRITE, or DELETE operation takes. However, multiple keys have little effect on READ timing and provide multiple access paths. Thus, they are most useful for files that are not subject to frequent additions and updates and are accessed in many different programs.

2. Select bucket sizes that reflect anticipated file activity and provide a suitable depth of index structure. (See Section 13.1.3.3, Index Depth.)

3. Avoid excessive duplication of key values. COBOL does not allow duplicates on the primary key, but permits them on alternate keys.

The following subsections deal with the specifics of indexed file design and creation.

13.1.3.2 Bucket Size - Bucket size selection can influence indexed file performance markedly.

To RMS, bucket size is expressed as an integral number of physical blocks, each 512 bytes long. Thus, a bucket size of 1 specifies a 512-byte bucket, while a bucket size of 2 specifies a 1024-byte bucket, and so on.

The COBOL compiler passes bucket size values to RMS based on what you specify in the BLOCK CONTAINS clause. There, you indicate bucket size in terms of records or characters, not physical blocks. As a COBOL user concerned with file optimization, you should be aware of the mechanism by which COBOL record and file descriptions are used to derive bucket sizes, so that you can predict how RMS will treat your file description.

If you express block size in records, the bucket can in some cases contain more records than you specify, but never fewer. For example, assume that your file contains fixed-length 100-byte records, and you call for each bucket to contain five records, as follows:

```
BLOCK CONTAINS 5 RECORDS
```

This might seem to define a bucket as a 512-byte block containing five records of 100 bytes each. However, the compiler adds RMS record and bucket overhead to each bucket for control purposes, as follows:

```
Bucket Overhead      15 bytes per bucket

Record Overhead       7 bytes per record (fixed-length)
                      9 bytes per record (variable-length)
```

Thus, in the example, bucket size is calculated as follows:

```
Bucket Overhead                              15 bytes
Record Size is 100 bytes
   + 7 bytes Record Overhead
       for each of 5 records
Total Record Space is (100 + 7)*5, or       535 bytes

Total Block specified by user               550 bytes
```

Because physical blocks are 512 bytes long, and buckets are always some integral number of physical blocks, the smallest buffer possible (the RMS default) in this case is two physical blocks (1024 bytes).

RMS, however, is not keyed to the BLOCK CONTAINS clause from which this bucket specification was derived, and puts as many records as will fit into each bucket. The bucket actually will contain nine records, not five.

The CHARACTERS option of the BLOCK CONTAINS clause allows you to specify bucket size more directly. For example:

```
BLOCK CONTAINS 2048 CHARACTERS
```

This calls for a bucket size of four 512-byte physical blocks. The number of characters in a bucket is always a multiple of 512. If you specify a value that is not a multiple of 512, RMS rounds it to the next higher multiple of 512.

13.1.3.3 Index Depth - The size of data records, key fields, and buckets in the file determines the depth of the index. Index depth, in turn, determines the number of disk accesses required to retrieve a particular record.

In general, performance is best with an index depth of 3 or 4. A shallower index will require fewer accesses, but will reduce available address space because of the larger buffers required.

13.1.3.4 Overhead Accumulation - In selecting a bucket size, you should consider the likely frequency of random insert and delete operations.

When a record is inserted, there must be sufficient room in the bucket to contain it. Otherwise, a bucket split occurs. Bucket splits can cause accumulation of storage overhead, thereby reducing usable space. The new bucket contains records moved from the original bucket (see Section 13.1.3) to make room for the new record. For each record moved out of the original bucket, a seven-byte pointer to the new location for that record remains in the original bucket. Thus, a bucket could accumulate overhead from bucket splits, possibly reducing usable space so much that it can no longer receive record insertions.

Record deletions also can accumulate storage overhead. Under most circumstances, however, most of the space that was occupied by the original record becomes available for reuse. Because duplicate primary keys are not allowed, RMS can reclaim all but two bytes of the deleted record space. This two-byte field is a flag indicating that a record has been deleted.

There are several ways to deal with the problem of overhead accumulation. First, determine or estimate the frequency of certain operations. If, for example, you expect only 100 records of a 100,000 record file to be added or deleted in an average month, your data base is stable enough that you might decide to allow some wasted space from record additions and deletions.

However, if you expect frequent additions and deletions, try the following:

1. Choose a bucket size that allows for overhead accumulation, if possible. Avoid bucket sizes that are an exact or near multiple of your record size.

2. To optimize for record insertion performance (as opposed to space optimization), first define the file with a fill number (using the RMS DEFINE utility or a MACRO program). A fill number specifies the number of bytes in the buckets of the file that you want to contain record information when the file is populated. Then, populate the file, specifying the /LO switch (see Table 6-2 or RMS utilities documentation). Thereafter, the unused space is available for record insertions, with minimum bucket splitting. Make certain that programs performing such record insertions do not specify the /LO switch.

13.2 OPTIMIZING PROGRAM ORGANIZATION

Program organization can influence I/O performance greatly. This section suggests guidelines toward an efficient program structure.

13.2.1 Sequential Reading of Indexed Files

If you access an indexed file sequentially, and the file is write-shared (using the /SH switch), performance improves if you use OPEN I-O instead of OPEN INPUT. Using OPEN I-O implies a possibility that you will write to the file--even though you have no intention of doing so.

Reading from a file that is open for input-output improves performance by locking the bucket, allowing you to obtain subsequent records from the same bucket without rereading it.

13.2.2 Caching Index Roots

RMS requires at least two buffers to process an indexed file. Each buffer is large enough to contain a single bucket. If your COBOL program does not contain a RESERVE n AREAS clause, the compiler allows RMS to set the default.

By including a RESERVE n AREAS clause in the SELECT statement for a file, you can create additional (but not fewer) buffers for the processing of an indexed file. At run time, RMS will retain (cache) the roots of one or more indexes of the file in memory. The random access of any record through that index will then require one less I/O operation.

The following rules apply for caching index roots:

1. The file must not be shared at run time.

2. Allocate one buffer for each key that your program uses to access file records, in addition to the two required buffers. For example, if the file contains a primary key and two alternate keys, and you use all of these keys to access records, allocate a total of five buffers. If you use only one key to access this file in a program, you need only one additional buffer area, or three in all.

3. Use the RESERVE n AREAS clause to obtain this allocation, where n is two more than the number of distinct keys used for access. For example, the clause RESERVE 5 AREAS causes allocation of the two required buffers, plus three buffer areas for caching the roots of three distinct file access keys.

13.2.3 Multi-block Reading and Writing

The multi-block read and write facility applies only to sequential files on disk devices. It allows reading or writing of more than one 512-byte block at a time during a single I/O operation, reducing the number of I/O operations needed to process a file. However, the single buffer used to process the file must be correspondingly longer.

To use this facility, be sure the file has SEQUENTIAL organization and resides on disk. Then, in the FD entry for the file, specify:

```
BLOCK CONTAINS n CHARACTERS
```

where n is a multiple of 512. Each multiple represents the number of physical blocks to be read or written during each access of the file. If n is not a multiple of 512, the compiler rounds the size to the next multiple of 512.

13.3 OPTIMIZING COMPUTATION

On arithmetic (ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE) and data movement (MOVE) operations, the compiler generates more efficient, specialized code if:

1. The data items involved in the computation (including the receiving fields) have the same type and scaling.

2. You omit the ROUNDED and SIZE ERROR phrases.

3. LINKAGE SECTION data is not involved in the computation.

Also, in the case of COMPUTATIONAL data, the data items must be the same size. Otherwise, the compiler uses slower, generalized code.

Certain data types allow faster computation than others. The data types, in order of most efficient to least efficient, are:

```
Signed COMPUTATIONAL
Unsigned COMPUTATIONAL
COMPUTATIONAL-3
DISPLAY
```

COMPUTATIONAL data items can be one, two, or four words long (see Chapter 4). To increase the efficiency of the generated code, define COMPUTATIONAL items with the same size; if necessary, make some items larger than you otherwise would. COMPUTATIONAL data items can have different PICTURE specifications and still be the same size. Items with PIC 9(1) to 9(4) are 1-word binary; with PIC 9(5) to 9(9), 2-word binary; and with PIC 9(10) to 9(18), 4-word binary.

On COMPUTATIONAL-3 (packed decimal) or DISPLAY data, operations are most efficient when performed on data items of minimal size. Different data size does not force the use of generalized code with these data types.

The following situations also force the use of generalized code, as opposed to more efficient code:

1. Non-matching decimal point alignment among the operands and receiving fields.

2. Use of the ROUNDED or SIZE ERROR options.

3. Data items defined in the LINKAGE SECTION of the source program.

Generalized code is necessary on LINKAGE SECTION data items, because their addresses are not known at compile time.

The following example illustrates the difference in execution time between specialized ADD code and the generalized ADD code.

```
01   A   PIC S9(4) USAGE COMP.
01   B   PIC S9(4) USAGE COMP.
01   C   PIC S9(4) USAGE COMP.
01   E   PIC S9(4)V9 USAGE COMP.
```

Of the following two ADD statements, statement (1) typically executes 30 to 40 times faster than statement (2):

```
(1)  ADD A B GIVING C.

(2)  ADD A B GIVING E.
```

On MULTIPLY and DIVIDE operations, decimal point alignment has a different meaning than for ADD and SUBTRACT operations. Assuming that the data types are the same for all items involved, the compiler uses the more efficient code if:

1. On a MULTIPLY, the product field scale factor equals the sum of the scale factors of the multiplicand and multiplier. For example:

```
01 X PIC S9(4)V9(2) USAGE COMP.
01 Y PIC S9(4)V9(3) USAGE COMP.
01 Z PIC S9(6)V9(5) USAGE COMP.

.
.
.

MULTIPLY X Y GIVING Z.
```

2. On a DIVIDE operation, the quotient scale factor equals the dividend scale factor minus the divisor scale factor. For example, using the data descriptions from the previous example:

```
DIVIDE Z BY X GIVING Y.
```

When defining data to be used as subscripts, 1-word signed COMPUTATIONAL is the most efficient. Try to avoid referencing tables by indexes unless you need to perform relative index references.

The use of arithmetic expressions increases use of temporary storage. It also generates larger operands, and can cause the less-efficient generalized code to be used unnecessarily. Avoid using the COMPUTE verb, and avoid using arithmetic expressions when specifying relational conditions.

APPENDIX A

THE COBOL FORMAT

COBOL NOTATION USED IN FORMATS

- Underlined upper-case words (key words) - required words;
- Upper-case words (not underlined) - optional words;
- Lower-case words - generic terms, must be supplied by the user;
- Brackets [] - enclosed portion is optional; if several enclosed words are vertically stacked, only one of them may be used;
- Braces {} - a selection must be made from the vertical stack of enclosed words;
- Ellipsis ... - the position at which repetition may occur;
- Comma and semicolon - optional punctuation;
- Period - required where shown in the formats.

NOTE: Shaded items represent PDP-11 COBOL extensions to the ANS-74 list of COBOL formats.

IDENTIFICATION DIVISION.

PROGRAM-ID. program-name.
[AUTHOR. [comment-entry]...]
[INSTALLATION. [comment-entry]...]
[DATE-WRITTEN. [comment-entry]...]
[DATE-COMPILED. [comment-entry]...]
[SECURITY. [comment-entry]...]

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. VAX-11.

OBJECT-COMPUTER. VAX-11 [MEMORY SIZE integer {WORDS / CHARACTERS / MODULES}]
[PROGRAM COLLATING SEQUENCE IS alphabet-name]
[SEGMENT-LIMIT IS segment-number].

```
[SPECIAL-NAMES.

  [CARD-READER IS mnemonic-name-1]
  [CONSOLE IS mnemonic-name-2]
  [LINE-PRINTER IS mnemonic-name-3]
  [PAPER-TAPE-PUNCH IS mnemonic-name-4]
  [PAPER-TAPE-READER IS mnemonic-name-5]

 [SWITCH integer-1   { ON STATUS IS condition-name-1    [OFF STATUS IS condition-name-2] }]
 [                   { OFF STATUS IS condition-name-2   [ON STATUS IS condition-name-1] }] ...
 [Alphabet-name IS  { NATIVE       }]
 [                  { STANDARD-1   }]
  [CURRENCY SIGN IS literal-1]
  [DECIMAL-POINT IS COMMA ].]

[INPUT-OUTPUT SECTION.

FILE-CONTROL. {file-control-entry}...
```

Format 1:

```
SELECT [OPTIONAL] file-name

     ASSIGN TO literal-1

     [, RESERVE integer-1      [AREA ]]
                               [AREAS]

     [; ORGANIZATION IS SEQUENTIAL]
     [; ACCESS MODE IS SEQUENTIAL]
     [; FILE STATUS IS data-name-1] .
```

Format 2:

```
SELECT file-name

     ASSIGN TO literal-1
     [, RESERVE integer-1      [AREA ]]
                               [AREAS]

      , ORGANIZATION IS RELATIVE

     [, ACCESS MODE IS  { SEQUENTIAL   [, RELATIVE KEY IS data-name-1] }]
                        { RANDOM       }                                 }]
                        { DYNAMIC      }  RELATIVE KEY IS data-name-1    }]

     [; FILE STATUS IS data-name-2] .
```

Format 3:

```
SELECT file-name

     ASSIGN TO literal-1

     [; RESERVE integer-1      [AREA ]]
                               [AREAS]

      ; ORGANIZATION IS INDEXED

     [; ACCESS MODE IS    {SEQUENTIAL}]
                          {RANDOM    }
                          {DYNAMIC   }

      ; RECORD KEY IS data-name-1
     [; ALTERNATE RECORD KEY IS data-name-2 [WITH DUPLICATES]]...
     [; FILE STATUS IS data-name-3] .

[I-O-CONTROL.
        [SAME [RECORD] AREA FOR file-name-1 {file-name-2}...]...
        [MULTIPLE FILE TAPE CONTAINS file-name-3 [POSITION integer-1]
          [file-name-4 [POSITION integer-2]...]...

     [APPLY      PRINT-CONTROL      ON file-name-5 [file-name-6]... ] ...].

DATA DIVISION.

[FILE SECTION.
[FD file-name

     [BLOCK CONTAINS [integer-1 TO] integer-2        {RECORDS   }]
                                                     {CHARACTERS}
     [RECORD CONTAINS [integer-3 TO]    integer-4    CHARACTERS]
      LABEL  {RECORD IS  }      {STANDARD}
             {RECORDS ARE}      {OMITTED }
     [VALUE OF ID IS {data-name-1}]
                     {literal-1  }
     [DATA  {RECORD IS  }      data-name-3    [data-name-4] ... ] ...
            {RECORDS ARE}
     [LINAGE IS      {data-name-5}    LINES     [WITH FOOTING AT   {data-name-6}]
                     {integer-5  }                                 {integer-6  }]

       [LINES AT TOP     {data-name-7}]  [LINES AT BOTTOM    {data-name-8}]]
                         {integer-7  }                       {integer-8  }

    [CODE-SET IS alphabet-name].
[record-description-entry]...]...]
[WORKING-STORAGE SECTION.
     [77-level-description-entry]
     [record-description-entry ]     ...]
```

```
[LINKAGE SECTION.
     [ 77-level-description-entry ]
     [ record-description-entry   ]...]
Data description entry:
Format 1:
     level-number { data-name-1 }
                  { FILLER      }
     [REDEFINES data-name-2]
[ { PICTURE } IS character-string ]
  { PIC     }
[                                                    ]
[   [USAGE IS] { COMPUTATIONAL   }                   ]
[              { COMP            }                   ]
[              { COMPUTATIONAL-3 }                   ]
[              { COMP-3          }                   ]
[              { DISPLAY         }                   ]
[              { DISPLAY-6       }                   ]
[              { DISPLAY-7       }                   ]
[              { INDEX           }                   ]
[                                                    ]
[ [SIGN IS] { LEADING  } [SEPARATE CHARACTER]        ]
[           { TRAILING }                             ]
[ { SYNCHRONIZED } [ { LEFT  } ]                     ]
[ { SYNC         } [ { RIGHT } ]                     ]
[ { JUSTIFIED }   [ { LEFT  } ]                      ]
[ { JUST      }   [ { RIGHT } ]                      ]
     [BLANK WHEN ZERO]
     [VALUE IS literal]
[ OCCURS { integer-1 TO integer-2 TIMES DEPENDING ON data-name-3 }
         { integer-2 TIMES                                       }
     [ { ASCENDING  } KEY IS data-name-4 [data-name-5]... ] ...
     [ { DESCENDING }                                    ]
       [INDEXED BY index-name-1 [index-name-2] ...] ].
Format 2:
  66 data-name-1 RENAMES data-name-2
     [ { THROUGH } data-name-3 ].
     [ { THRU    }             ]
Format 3:
  88 condition-name { VALUE IS  } literal-1 [ { THROUGH } literal-2 ]
                    { VALUES ARE }           [ { THRU    }         ]
     [literal-3 [ { THROUGH } literal-4 ]] ... .
                [ { THRU    }         ]

PROCEDURE DIVISION [USING [data-name-1][,data-name-2] ...].
```

Format 1:

```
[DECLARATIVES.
[section-name SECTION [segment-number] . declarative-sentence
[paragraph-name.[sentence]...]...}...
END DECLARATIVES.]
{section-name SECTION [segment-number].
[paragraph-name.[sentence]...]...}...
```

Format 2:

```
[paragraph-name.[sentence]...}...
```

STATEMENTS

```
ACCEPT identifier [FROM mnemonic-name]

ACCEPT identifier FROM  { DATE }
                        { DAY  }
                        { TIME }

ADD   { identifier-1 }  [ identifier-2 ]    ... TO identifier-m [ROUNDED]
      { literal-1    }  [ literal-2    ]
      [identifier-n [ROUNDED]]...[ON SIZE ERROR imperative-statement]

ADD   { identifier-1 }  { identifier-2 }  [ identifier-3 ]    ...
      { literal-1    }  { literal-2    }  [ literal-3    ]
   GIVING identifier-m [ROUNDED]   [identifier-n [ROUNDED]]  ...
          [ON SIZE ERROR imperative-statement]

ADD   { CORRESPONDING }
      { CORR          }          identifier-1 TO identifier-2 [ROUNDED]
       [ON SIZE ERROR imperative-statement]

ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2
   [procedure-name-3 TO [PROCEED TO] procedure-name-4]...

CALL literal

[ USING   { {[BY REFERENCE]}                                          }
          { { BY VALUE     }  identifier-1  [identifier-2] ...        }
          { { BY DESCRIPTOR}                                          }
  [ { BY REFERENCE }                                                  ]
  [ { BY VALUE     }  identifier-3  [identifier-4] ...                ] ... ]
  { BY DESCRIPTOR }                                                   ]

  [GIVING identifier-5]

CLOSE file-name-1  [ { REEL }  [WITH NO REWIND]                ]  [ file-name-2  [ { REEL }  [WITH NO REWIND]                ] ...
                   { UNIT }  [FOR REMOVAL     ]                ]                  [ { UNIT }  [FOR REMOVAL     ]                ]
                   WITH { NO REWIND }                          ]                  [ WITH { NO REWIND }                          ]
                        { LOCK      }                          ]                  [      { LOCK      }                          ]
```

```
COMPUTE identifier-1 [ROUNDED] [identifier-2 [ROUNDED]] ...
  = arithmetic-expression [ON SIZE ERROR imperative-statement]

DELETE file-name RECORD    [INVALID KEY imperative-statement]
DISPLAY {identifier-1} [{identifier-2}]  ...
        {literal-1   }  {literal-2   }
  [UPON mnemonic-name] [WITH NO ADVANCING]

DIVIDE  {identifier-1}     INTO identifier-2 [ROUNDED]
        {literal-1   }
  [identifier-3[ROUNDED]] ... [ON SIZE ERROR imperative-statement]

DIVIDE  {identifier-1}     INTO  {identifier-2}     GIVING identifier-3[ROUNDED]
        {literal-1   }           {literal-2   }
  [identifier-4[ROUNDED]]...[ON SIZE ERROR imperative-statement]

DIVIDE  {identifier-1}     BY    {identifier-2}     GIVING identifier-3[ROUNDED]
        {literal-1   }           {literal-2   }
  [identifier-4[ROUNDED]]...[ON SIZE ERROR imperative-statement]

DIVIDE  {identifier-1}     INTO  {identifier-2}     GIVING identifier-3[ROUNDED]
        {literal-1   }           {literal-2   }
  REMAINDER identifier-4[ON SIZE ERROR imperative-statement]

DIVIDE  {identifier-1}     BY    {identifier-2}     GIVING identifier-3[ROUNDED]
        {literal-1   }           {literal-2   }
  REMAINDER identifier-4[ON SIZE ERROR imperative-statement]

EXIT [PROGRAM]

GO TO [procedure-name-1]
GO TO procedure-name-1 [procedure-name-2]...procedure-name-n DEPENDING ON identifier

IF condition  {statement-1  }  [ELSE statement-2]
              {NEXT SENTENCE}  [ELSE NEXT SENTENCE]

INSPECT identifier-1 TALLYING
    {identifier-2 FOR  {{ALL      }  {identifier-3}  [{BEFORE}  INITIAL  {identifier-4}]} ...}
                       {{LEADING  }  {literal-1   }  [{AFTER }          {literal-4   }]}
                       {CHARACTERS}                                                    } ...

INSPECT identifier-1 REPLACING
    {CHARACTERS BY  {identifier-6}  [{BEFORE}  INITIAL  {identifier-7}]                                                  }
                    {literal-4   }  [{AFTER }          {literal-5   }]                                                  }
    {{{ALL      }  {identifier-5}  BY  {identifier-8}  [{BEFORE}  INITIAL  {identifier-9}]} ...}                        } ...
    {{{LEADING  }  {literal-3   }      {literal-4   }  [{AFTER }          {literal-5   }]}                             }
    {{{FIRST    }                                                                        }                             }

INSPECT identifier-1 TALLYING
    {identifier-2 FOR  {{ALL       }  {identifier-3}  [{BEFORE}  INITIAL  {identifier-4}]} ...}
                       {{LEADING   }  {literal-1   }  [{AFTER }          {literal-2   }]}
                       {CHARACTERS}                                                    } ...

REPLACING
    {CHARACTERS BY  {identifier-6}  [{BEFORE}  INITIAL  {identifier-7}]                                                  }
                    {literal-4   }  [{AFTER }          {literal-5   }]                                                  }
    {{{ALL      }  {identifier-5}  BY  {identifier-8}  [{BEFORE}  INITIAL  {identifier-9}]} ...}                        } ...
    {{{LEADING  }  {literal-3   }      {literal-4   }  [{AFTER }          {literal-5   }]}                             }
    {{{FIRST    }                                                                        }                             }
```

```
MOVE    {identifier-1}
        {literal     }   TO identifier-2 [identifier-3] ...

MOVE    {CORRESPONDING}
        {CORR         }   identifier-1 TO identifier-2

MULTIPLY {identifier-1}
         {literal-1   }   BY identifier-2 [ROUNDED]
    [identifier-3 [ROUNDED]] ... [ON SIZE ERROR imperative-statement]

MULTIPLY {identifier-1}   BY   {identifier-2}   GIVING identifier-3 [ROUNDED]
         {literal-1   }        {literal-2   }
    [identifier-4 [ROUNDED]] ... [ON SIZE ERROR imperative-statement]

OPEN    {INPUT file-name-1[WITH NO REWIND] [file-name-2 [WITH NO REWIND]]...  }
        {OUTPUT file-name-3[WITH NO REWIND] [file-name-4 [WITH NO REWIND]]...}   ...
        {I-O file-name-5 [file-name-6]...                                    }
        {EXTEND file-name-7 [file-name-8]...                                 }

PERFORM procedure-name-1   [{THROUGH}  procedure-name-2]
                           {THRU   }

PERFORM procedure-name-1   [{THROUGH}  procedure-name-2]   {identifier-1}   TIMES
                           {THRU   }                       {integer-1   }

PERFORM procedure-name-1   [{THROUGH}  procedure-name-2]   UNTIL condition-1
                           {THRU   }

PERFORM procedure-name-1   [{THROUGH}  procedure-name-2]
                           {THRU   }

    VARYING {identifier-2}        FROM {identifier-3}
            {index-name-1}             {index-name-2}
                                       {literal-1   }

         BY {identifier-4}        UNTIL condition-1
            {literal-2   }

[   AFTER   {identifier-5}        FROM {identifier-6}
            {index-name-3}             {index-name-4}
                                       {literal-3   }

         BY {identifier-7}        UNTIL condition-2
            {literal-4   }

[   AFTER   {identifier-8}        FROM {identifier-9}
            {index-name-5}             {index-name-6}
                                       {literal-5   }

         BY {identifier-10}       UNTIL condition-3]]
            {literal-6   }

READ file-name [NEXT] RECORD [INTO identifier] [AT END imperative-statement]
READ file-name RECORD [INTO identifier] [INVALID KEY imperative-statement]
READ file-name RECORD [INTO identifier] [KEY IS data-name] [INVALID KEY imperative-statement]
REWRITE record-name [FROM identifier] [INVALID KEY imperative-statement]
```

```
SEARCH identifier-1  [VARYING  {identifier-2}]  [AT END imperative-statement-1]
                               {index-name-1 }

     WHEN condition-1  {imperative-statement-2}
                       {NEXT SENTENCE        }

    [WHEN condition-2  {imperative-statement-3}]  ...
                       {NEXT SENTENCE        }

SEARCH ALL identifier-1[AT END imperative-statement-1]

          {data-name-1     }  {IS EQUAL TO}  {identifier-3                }
     WHEN {                }  {IS =      }  {literal-1                   }
          {condition-name-1}                 {arithmetic-expression-1    }

          [AND  {data-name-2     }  {IS EQUAL TO}  {identifier-4                }  ...]
                {condition-name-3}  {IS =      }  {literal-2                   }
                                                  {arithmetic-expression-2    }

          {imperative-statement-2}
          {NEXT SENTENCE         }

SET  {identifier-1  [identifier-2]...}  TO  {identifier-3}
     {index-name-1  [index-name-2]...}      {index-name-3}
                                            {integer-1   }

SET  index-name-4  [index-name-5]...  {UP BY  }  {identifier-4}
                                      {DOWN BY}  {integer-2   }

START file-name  [KEY  {IS EQUAL TO                }  data-name]
                       {IS =                       }
                       {IS GREATER THAN            }
                       {IS >                       }
                       {IS NOT LESS THAN           }
                       {IS NOT <                   }
     [INVALID KEY imperative-statement]

STOP  {RUN    }
      {literal}

STRING  {identifier-1}  [identifier-2]  ...  DELIMITED BY  {identifier-3}
        {literal-1   }  [literal-2   ]                      {literal-3   }
                                                            {SIZE        }

     [{identifier-4}  [identifier-5]  ...  DELIMITED BY  {identifier-6}]  ...
      {literal-4   }  [literal-5   ]                     {literal-6   }
                                                         {SIZE        }
       INTO identifier-7 [WITH POINTER identifier-8]
            [ON OVERFLOW imperative-statement]

SUBTRACT  {identifier-1}  [identifier-2]  ...  FROM identifier-m[ROUNDED]
          {literal-1   }  [literal-2   ]
    [identifier-n[ROUNDED]],...[ON SIZE ERROR imperative-statement]
```

```
SUBTRACT   { identifier-1 }   [ identifier-2 ]   ...   FROM   { identifier-m }
           { literal-1    }   [ literal-2    ]              { literal-m    }

    GIVING identifier-n[ROUNDED] [identifier-o[ROUNDED]]...
    [ON SIZE ERROR imperative-statement]
SUBTRACT   { CORRESPONDING }   identifier-1 FROM identifier-2 [ROUNDED]
           { CORR          }
    [ON SIZE ERROR imperative-statement]

UNSTRING identifier-1
    [ DELIMITED BY [ALL]   { identifier-2 }   [ OR [ALL]   { identifier-3 } ] ... ]
                           { literal-1    }                { literal-2    }
    INTO identifier-4[DELIMITER IN identifier-5] [COUNT IN identifier-6]
    [identifier-7 [DELIMITER IN identifier-8][COUNT IN identifier-9]]...
    [WITH POINTER identifier-10][TALLYING IN identifier-11]
    [ON OVERFLOW imperative-statement]

USE AFTER STANDARD   { EXCEPTION }   PROCEDURE ON   { file-name-1[file-name-2]... }
                     { ERROR     }                  { INPUT                       }
                                                    { OUTPUT                      }
                                                    { I-O                         }
                                                    { EXTEND                      }

WRITE record-name[FROM identifier-1]
    [ { BEFORE }   ADVANCING   { { { identifier-2 }   [ LINE  ] } }
      { AFTER  }               { { { integer    }   [ LINES ] } }
                               { [PAGE]                        }  ]
    [AT   { END-OF-PAGE }   imperative-statement]
          { EOP         }

WRITE record-name [FROM identifier] [INVALID KEY imperative-statement]
```

```
COPY   { text-name }
       { literal-3 }
    [ REPLACING   { { literal-1 }   BY   { literal-2 } } ... ]
                  { { word-1    }        { word-2    } }
```

NOTE: A COPY statement may appear anywhere that a word appears in the COBOL source program.

APPENDIX B

COMPILER IMPLEMENTATION LIMITATIONS

This appendix describes the implementation limitations for the VAX-11 COBOL-74 compiler system (compiler and RTS). You should not confuse the term "limitation" with "restriction". A restriction is a language facility that is not implemented or should not be used due to known errors in its implementation. An implementation limitation quantifies the limits of a language facility that is supported by the system.

Practical implementation limitations exist in every compiler; They result from the finite size of compiler tables, compiler data structure representations, and so on. Since the VAX-11 COBOL-74 compiler employs a Virtual Memory System to support many compiler data structures, the quantities specified for some implementation limitations are approximations. However, as a general rule, the following guidelines should not be exceeded in the development of a COBOL program.

IMPLEMENTATION LIMITATIONS

1. The Data Division of a COBOL program cannot be larger than 65K bytes. A file description entry cannot be larger than 32K bytes.

2. A DISPLAY statement cannot contain more than 16 sending operands.

3. The maximum number of data-name definitions in a COBOL program is approximately 2000.

4. The maximum number of procedure-name definitions in a COBOL program is approximately 2000.

5. Because file description level-numbers can range from 01 to 49, level 88 condition-names can have no more than 50 qualifiers (FD through 49). Data-names declared in the File Section can have no more than 49 qualifiers (FD through 48). Data-names declared in the Working-Storage and Linkage Sections can have no more than 48 qualifers (01 - 48).

6. A GO TO DEPENDING statement can have no more than 16 operands.

APPENDIX C

SOURCE PROGRAM LISTINGS

This appendix contains compiler listings for two COBOL programs. The first, STATB, calls three subprograms; the second, DOCATS, is one of the subprograms.

The examples demonstrate some of the features of VAX-11 COBOL-74, such as:

- The COPY statement
- The COPY REPLACING statement
- The CALL statement
- The results of using the /MAP and /VERB_LOCATION compiler qualifiers

The circled numbers on the source listings indicate features that are annotated in the text.

Source Listing Features

(1)- The version of the VAX-11 COBOL-74 compiler.

(2)- The source file, including file type, or extension, and version number.

(3)- Date and time when the compilation began.

(4)- The compiler command line. The contents of the command line can help to explain why the listing looks like it does and how the program runs. For example, this command line shows that the /VERB_LOCATION and /MAP qualifiers were used.

(5) - The IDENTification number assigned by the compiler. This number identifies the specific compilation of the program and is used as an additional identifier for the object module.

(6) - Source line number assigned by the compiler. This number is used in RTS error message displays to indicate the location at which the error was detected. It also appears in error message displays that show nested PERFORMs.

(7) - Sequence number. If the source file used conventional format (/ANSI_FORMAT), the sequence field (positions 1-6) appears here.

(8) - Source text. This area contains the text that was processed by the compiler. If a line of text was too long, only the part that appears here was processed. The compiler also prints a diagnostic message when it truncates a line of source text.

(9) - Identification field. If the source file used conventional format, this area would contain the identification field (positions 73-80).

(10) - Identifies a source line that: a) contains a COPY statement, or b) was copied from a library file.

(11) - COBOL verb (appears only when /VERB_LOCATION qualifier is used). Identifies the COBOL verb that is referred to by the other entries on the line.

(12) - Segment number (/VERB_LOCATION qualifier only). Identifies the program segment, or PSECT. Notice that this is not the PSECT name; it is a consecutive number assigned to all procedural PSECTS during compilation and duplicates the segment numbers in other programs.

(13) - Offset (/VERB_LOCATION qualifier only). Specifies the hexadecimal offset (distance) from the beginning of the segment for the object code generated by the COBOL verb (number 11).

(14) - Offset (/VERB_LOCATION qualifier only). Specifies the hexadecimal offset in hexidecimal bytes from the beginning of the program entry-point (STATE).

(15) - Compiler diagnostic severity code. Describes the seriousness of the compiler diagnostic. This diagnostic is "informational", which means that the compiler can take corrective action.

(16) - Diagnostic source line number. Identifies the source line to which the diagnostic applies. In this case, OPTIONS-AREA is defined as larger than CUSTOMER-FILE-ID; therefore, truncation occurs.

(17) - Compiler diagnostic number. Identifies the specific diagnostic. Use this number to find a description of the diagnostic in Appendix D.

(18) - Diagnostic message. A one-line description of the condition.

(19) - Data Map. Describes the data-names and file-names used in the program. This section appears only if the /MAP qualifier is used.

(20) - Level. Contains the level-indicator or level-number of the item. An L preceding the level indicates that the data-name is a Linkage Section item.

(21) - Name. The file-name or data-name.

(22) - Source line. The file-name or data-name is defined on this source line in the Data Division.

(23) - Data Division location. Identifies the hexadecimal offset of the file or data-name from the beginning of data PSECT. For Linkage Section data-names, the offset is from the 01-level.

(24) - Directory location. Identifies the hexadecimal offset of the data item's descriptor. For Linkage Section data items, the offset is from the 01-level. The RTS uses the descriptor to operate on a data item.

A directory location that contains asterisks indicates that the compiler did not generate a descriptor because the data-name was not used in the Procedure Division.

(25) - USAGE. Corresponds to the USAGE clause or implicit usage of the data item description. The following abbreviations are used:

DISP	DISPLAY
CMP	COMPUTATIONAL
CMP3	COMPUTATIONAL-3
INDX	INDEX

(26) - Class. Identifies the COBOL class of the data item. The compiler determines class from the PICTURE or level associated with the data-name. The following abbreviations are used:

ALPHA	Alphabetic
NUM	Numeric
AN	Alphanumeric
ANEDIT	Alphanumeric Edited
NMEDIT	Numeric Edited

(27) - Occurrence level. Indicates the number of subscripts necessary to refer to the data-name.

(28) - Length. Specifies the length of the data item in decimal bytes.

(29) - Procedure Name Map. Describes the procedure-names that appear in the program. This section appears only if the /MAP qualifier is used.

(30) - Procedure-name. This is the name as it appears in the Procedure Division.

(31) - Source line. Identifies the source line in which the procedure-name is defined.

(32) - PSECT. Identifies the name of the executable code PSECT (program section) in which the procedure-name appears. PSECT name consists of the first 11 characters of PROGRAM-ID (padded on the right by "$" if less than 11), followed by a three-digit number.

(33) - Offset. Specifies the hexadecimal offset (distance) of the location of the procedure-name from the beginning of the PSECT.

(34) - Segment-number. Corresponds to the segment-number in the header for the section in which the procedure-name appears.

(35) - Section. An "S" indicates that the procedure-name is a section-name.

(36) - Paragraph. A "P" indicates that the procedure-name is a paragraph-name.

(37) - Segmentation Map. Describes the segmentation for each Procedure Division section. This map appears only when the /MAP qualifier is used.

(38) - Section Name. The name of the section as it appears in the Procedure Division.

(39) - Segment-number. The segment-number specified in the section header, or the implied segment-number 00.

(40) - PSECT Name. Indicates the name of the procedural PSECT generated for the section. If the generated code exceeds the code segment limit, the compiler generates additional PSECTs; their names are displayed beneath the first.

(41) - The size of the procedural PSECT in hexadecimal bytes.

(42) - The size of the procedural PSECT in decimal bytes.

(43) - Compiler-Generated PSECTs. Describes the procedural PSECT's generated by the compiler (/MAP qualifier) to provide run-time execution initialization.

(44) - PSECT name.

(45) - The size of the PSECT in hexadecimal bytes.

(46) - The size of the PSECT in decimal bytes.

(47) - Referenced RTS Routines. Lists the names of all COBOL RTS routines (/MAP qualifier) that are referenced by the compiler-generated code. All RTS routines have the form: C74$<name>.

(48) - Data PSECT Map. Lists the nonexecutable PSECTs generated by the compiler (/MAP qualifier).

(49) - PSECT name.

(50) - The size of the PSECT in hexadecimal bytes.

(51) - The size of the PSECT in decimal bytes.

(52) - External Subprogram References. Lists the names of all subprograms (/MAP qualifier) referenced by CALL statements in the program.

(53) - Error Severity Code. Describes the seriousness of errors. Chapter 10 describes the severity codes and their meanings.

(54) - Error Count. The number of compilation errors detected for each severity level.

```
STATB                                                    12-Jan-1979 00:00:15    VAX-11 COBOL-74  V4.00-BL
IDENT: 012000                                                                    STATB  SUB:0

/NOLIST_FORMAT        /COPY_LIST            /ACCESS_REFERENCE  /LIST=STATB
/MAP                  /VERSION_CAPTION      /NAMES:ANSI        /OBJECT=STATB.OBJ
/DEBUG:ALL:NODEBUG
```

```
00001          IDENTIFICATION DIVISION.
00002          PROGRAM-ID.         STATB.
00003          AUTHOR.                         R FRIES.
00004          INSTALLATION.                   JONES MAIL ORDER COMPANY.
00005          DATE-WRITTEN.                   17 JANUARY 1979.
00006          DATE-COMPILED.
00007
00008          *            Using calling programs this program demonstrates
00009          *            the effects and advantages of modular program
00010          *            development.  Depending on operator-modified
00011          *            options and the contents of data records, the
00012          *            program generates various outputs.
00013          *
00014          *            The called programs are
00015          *
00016          *               NAME      FUNCTION
00017          *
00018          *              EXCEPT    Generates an exception report.
00019          *              OCCUPY    Generates mailing labels.
00020          *              CRED_LM   Generates "credit limit" letters.
00021          *
00022          *
00023          ****************************************************************
00024
00025               17-Jan-1979 ,
00026               ENVIRONMENT DIVISION.
00027
00028          ****************************************************************
00029
00030               CONFIGURATION SECTION.
00031
00032               SOURCE-COMPUTER.    VAX-11,
00033               OBJECT-COMPUTER.    VAX-11,
00034
00035          ****************************************************************
00036
00037               INPUT-OUTPUT SECTION.
00038
00039               FILE-CONTROL.
00040
00041                   SELECT CUSTOMER-FILE
00042                       ASSIGN TO "CUSTOM"
00043                       ORGANIZATION IS INDEXED
00044                       ACCESS MODE IS DYNAMIC
00045                       RECORD KEY IS CUST-CUST-NUMBER
00046                       ALTERNATE RECORD KEY IS CUST-CUSTOMER-NAME
00047                       FILE STATUS IS CUSTOMER-FILE-STATUS.
00048
00049                   SELECT STATEMENT-REPORT
00050                       ASSIGN TO "STATEM"
00051                       FILE STATUS IS STATEMENT-REPORT-STATUS.
00052
00053          ****************************************************************
00054
00055               DATA DIVISION.
00056
00057          ****************************************************************
00058
00059               FILE SECTION.
00060
00061               FD  CUSTOMER-FILE
00062                   LABEL RECORDS ARE STANDARD
00063                   VALUE OF ID IS CUSTOMER-FILE-ID.
00064
00065               COPY CUSTREC.
00066
00067
00068               01  CUSTOMER-FILE-RECORD.
00069                   05      CUST-CUST-NUMBER                PIC      X(6).
00070                   05      CUST-CUSTOMER-NAME              PIC      X(30).
00071                   05      CUST-ADDRESS-LINE-1             PIC      X(30).
00072                   05      CUST-ADDRESS-LINE-2             PIC      X(30).
00073                   05      CUST-ADDRESS-LINE-3             PIC      X(30).
00074                   05      CUST-ADDRESS-ZIP-CODE           PIC      X(5).
00075                   05      CUST-PHONE.
00076                       49      CUST-P-ONE-AREA-CODE        PIC      X(3).
00077                       49      CUST-P-ONE-EXCHANGE         PIC      X(3).
00078                       49      CUST-P-ONE-LAST-4           PIC      9(4).
00079                   05      CUST-PHONE-NO-REDEFINED
00080                           REDEFINES CUST-PHONE            PIC      9(10).
00081                   05      CUST-ATTENTION-LINE             PIC      X(22).
00082                   05      CUST-CREDIT-LIMIT               PIC      9(5)V99.
```

```
000083          05      CUST-HEADER-DATA REDEFINES CUST-CREDIT-LIMIT.
000084              10      FILLER                  PIC     X(4).
000085              10      NEXT-ACCT-NUMBER        PIC     9(4).
000086          05      CUST-DUE-AMT
000087                                              PIC     9(10)V99.
000088          05      CUST-WEIGHT
000089                                              PIC     9(11)V999.
000090          05      CUST-ORDER-SEQUENCE         PIC     9(4).
000091          05      CUST-PAYMENT-SEQUENCE       PIC     9(4).
000092
000093      FD  STATEMENT-REPORT
000094          LABEL RECORDS ARE STANDARD.
000095      01  STATEMENT-REC-OUT.
000096          05  FILLER                          PIC     X(5).
000097          05  ADDRESS-LINE-OUT                PIC     X(30).
000098          05  FILLER                          PIC     X(3).
000099          05  ADDRESS-ZIP                     PIC     X(5).
000100          05  FILLER                          PIC     X(35).
000101          05  FORM-NAME.
000102              10  FILLER                      PIC     X(7).
000103              10  FORM-DATE                   PIC     X(8).
000104
000105      01  S-R-R-2.
000106          05  FILLER                          PIC     X(151).
000107          05  REPORT-CREDIT                   PIC     Z,ZZZ,ZZZ.ZZ9,99.
000108          05  FILLER                          PIC     X(38).
000109          05  REPORT-PIC                      PIC     Z,ZZZ,ZZZ,ZZ9,99.
000110
000111      01  S-R-R-3.
000112          05  STATEMENT-DATE                  PIC     X(12).
000113          05  FILLER                          PIC     X(6).
000114          05  STATEMENT-CAPTION               PIC     X(93).
000115          05  STATEMENT-BALANCE               PIC     Z,ZZZ,ZZZ,ZZ9.99.
000116
000117  ********************************************************************
000118
000119      WORKING-STORAGE SECTION.
000120
000121      01  CUSTOMER-FILE-STATUS                PIC     X(2).
000122      01  STATEMENT-REPORT-STATUS             PIC     X(2).
000123      01  CUSTOMER-ID-IN                      PIC     X(10)
000124              VALUE "CUSTOM.DAT".
000125      01  TODAYS-DATE                         PIC     9(6).
000126      01  TODAYS-REDEFINES REDEFINES TODAYS-DATE.
000127          05  TODAY-YEAR                      PIC     9(2).
000128          05  TODAY-MONTH                     PIC     9(2).
000129          05  TODAY-DAY                       PIC     9(2).
000130      01  TODAYS-REPORT-DATE.
000131          05  TODAY-MONTH                     PIC     99.
000132          05  FILLER                          PIC     X(1) VALUE "/".
000133          05  TODAY-DAY                       PIC     9(2).
000134          05  FILLER                          PIC     X(1) VALUE "/".
000135          05  TODAY-YEAR                      PIC     9(2).
000136
000137      01  STATUS-MESSAGE                      PIC     X(50) VALUE SPACES.
000138
000139      01  DISP-RECORD.
000140          05  FILLER                          PIC     X(58)   VALUE SPACES.
000141          05  DISP-NUM                        PIC     Z(5).
000142
000143      01  YTD-DATA-DISPLAY                    PIC     9(10) VALUE ZEROS.
000144
000145      01  EXCEPTION-INDICATORS.
000146          05  EXCEPTION-INDICATOR OCCURS 10   PIC     9(1).
000147
000148      01  OPTIONS-ARG.
000149          05  OPTION-ARG-CHAR   OCCURS 10     PIC     X(1).
000150
000151      01  X-COUNT                             PIC     9(2).
000152
000153      01  OPTION-STORAGE.
000154          03  OPTION-ENTRY        OCCURS 5    PIC     9(1).
000155      01  OPTION-VALUES REDEFINES OPTION-STORAGE.
000156          03  FILLER                          PIC     9(1).
000157              88  WANT-STATEMENTS             VALUE 1 THRU 9.
000158          05  FILLER                          PIC     9(1).
000159              88  WANT-AVG-CUBS               VALUE 1 THRU 9.
000160          05  FILLER                          PIC     9(1).
000161              88  WANT-ALL-CATALOGS           VALUE 1 THRU 9.
000162          05  FILLER                          PIC     9(1).
000163              88  WANT-SHORT-CATALOGS         VALUE 1 THRU 9.
000164          05  FILLER                          PIC     9(1).
000165              88  WANT-CREDIT-LIMIT-LETTERS VALUE 1 THRU 9.
000166          03  FILLER                          PIC     X(3).
000167
000168      01  RECORD-COUNT                        PIC     9(3) VALUE 0.
000169      01  STATEMENT-COUNT                     PIC     9(5) VALUE 0.
000170      01  INVOICE-COUNT                       PIC     9(5) VALUE 0.
000171      01  CREDIT-LIMIT-COUNT                  PIC     9(5) VALUE 0.
```

```
                                    01  CATALOG-COUNT                    PIC     9(5) VALUE 0.

      *****************************************************************

       PROCEDURE DIVISION.

      *****************************************************************

       DECLARATIVES.

       CUSTOMER-ERROR SECTION.

           USE AFTER STANDARD ERROR PROCEDURE ON CUSTOMER-FILE.
       PARA-1.

       DISPLAY-1.
           DISPLAY "1-D BRANCH ON CUSTOMER-FILE, CODE ("
               CUSTOMER-FILE-STATUS
               ")".

       STOP-1.
           STOP RUN.

       STATEMENT-ERROR SECTION.

           USE AFTER STANDARD ERROR PROCEDURE ON STATEMENT-REPORT.
       BEGIN.

       DISPLAY-2.
           DISPLAY "1-C ERROR ON STATEMENT-REPORT, CODE ("
               STATEMENT-REPORT-STATUS
               ")".

       STOP-2.
           STOP RUN.

       END DECLARATIVES.

      *****************************************************************
      *
      *            This section performs housekeeping
      *            functions only.
      *
       START-UP-HOUSEKEEPING SECTION 99.
       BEGIN.

       ACCEPT-1.
           ACCEPT TODAYS-DATE FROM DATE.

       MOVE-1.
           MOVE CORRESPONDING TOR TO TODAYS-REPORT-DATE.

       MOVE-2.
           MOVE SPACES TO OPTIONS-AREA.

      *            Set CUSTOMER-FILE name. Use default
      *            if none is entered.

       DISPLAY-3.
           DISPLAY " ENTER CUSTOMER FILE NAME (OR CR)".

       ACCEPT-2.
           ACCEPT OPTIONS-AREA.

       IF-1.
           IF OPTIONS-AREA NOT = SPACES
               MOVE OPTIONS-AREA TO CUSTOMER-FILE-ID

(15) (16) (17) (18)
1 00217 NOTE: POSSIBLE HIGH ORDER TRUNCATING FIELD IDENTIFICATION.

       MOVE-3.
               MOVE SPACES TO OPTIONS-AREA.

      *            Set options from the operator and
      *            store results. Ignore non-standard
      *            option input.

       DISPLAY-4.
           DISPLAY " ENTER OPTIONS:".

       DISPLAY-5.
           DISPLAY "  B  = Print statements".

       DISPLAY-6.
           DISPLAY "  I  = Print invoices".

       DISPLAY-7.
           DISPLAY "  CA = Set all catalogs".

       DISPLAY-8.
           DISPLAY "  CC = Set selective catalogs".

       DISPLAY-9.
           DISPLAY "  CL = Create limit letters".

       ACCEPT-3.
           ACCEPT OPTIONS-AREA.

       MOVE-4.
           MOVE ALL ZERO TO OPTION-STORAGE.

       IF-2.
           IF OPTIONS-AREA = SPACES
               DISPLAY "Statements Report Only"
```

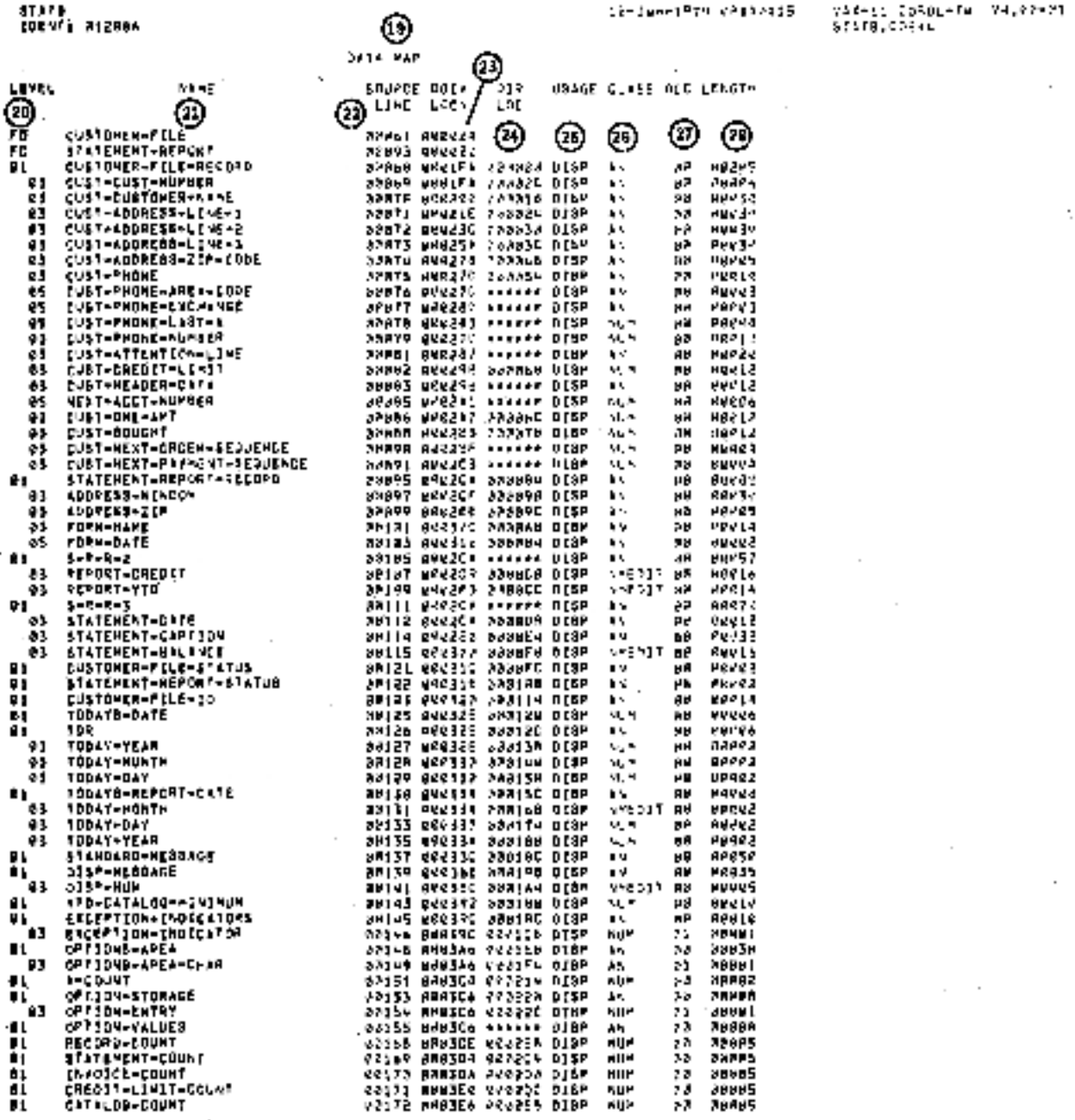
```
GO          : 03  000183  (2)4A02(1)
                          00250
                          00251          GO TO CONFIRM-OPTIONS.
MOVE        : 02  000187  (2)3A02(1)
                          00252
                          00253      MOVE 2 TO A-COUNT.
INSPECT     : 03  00018A  (3)3B02(2)
                          00254
                          00255      INSPECT OPTIONS-AREA TALLYING
                          00256          OPTION-ENTRY (1) FOR ALL "A"
                          00257          OPTION-ENTRY (2) FOR ALL "B"
                          00258          OPTION-ENTRY (3) FOR ALL "C"
                          00259          OPTION-ENTRY (4) FOR ALL "D"
                          00260          OPTION-ENTRY (5) FOR ALL "E".
                          00261
IF          : 03  000224  (4)4B02(2)
                          00262
                          00263      IF OPTION-STUPAGE = 'Y' GO TO
DISPLAY     : 03  000258  (2)3B02(2)
                          00264          DISPLAY 'No options recognized'
STOP        : 03  000270  (2)3B02(3)
                          00265          STOP RUN.
                          00266
DISPLAY     : 03  000284  (2)3B02(1)
                          00267      DISPLAY 'Selected options:'.
IF          : 03  000290  (2)3B02(2)
                          00268      IF WANT-STATEMENTS
DISPLAY     : 03  0002AC  (2)3B02(3)
                          00269          DISPLAY ' Statements'.
IF          : 03  0002C8  (2)3B02(4)
                          00270      IF WANT-INVOICES
DISPLAY     : 03  0002E4  (2)3B02(5)
                          00271          DISPLAY ' Invoices'.
IF          : 03  0002FC  (2)3B02(6)
                          00272      IF WANT-ALL-CATALOGS
DISPLAY     : 03  00031C  (2)3B02(7)
                          00273          DISPLAY ' All catalogs'.
IF          : 03  000334  (2)3B02(8)
                          00274      IF WANT-SO-E-CATALOGS
DISPLAY     : 03  000354  (2)3B03(1)
                          00275          DISPLAY ' Selected catalogs'.
IF          : 03  00036C  (2)3B03(2)
                          00276      IF WANT-CREDIT-LIMIT-LETTERS
DISPLAY     : 03  00038C  (2)3B03(3)
                          00277          DISPLAY ' Credit limit letters'.
                          00278
                          00279  CONFIRM-OPTIONS.
DISPLAY     : 03  0003AC  (2)3B03(4)
                          00280      DISPLAY 'CONFIRM OPTIONS (Y)es or (N)o'.
ACCEPT      : 03  0003C4  (2)3B03(5)
                          00281      ACCEPT OPTIONS-AREA.
IF          : 03  0003D4  (2)3B03(6)
                          00282      IF OPTION-AREA-CHAR (1) NOT = 'Y' AND 'N'
GO          : 03  0003E8  (2)3B03(7)
                          00283          GO TO CONFIRM-OPTIONS.
IF          : 03  0003F0  (2)3B03(8)
                          00284      IF OPTION-AREA-CHAR (1) = 'N'
DISPLAY     : 03  00043C  (2)3B04(1)
                          00285          DISPLAY 'ABORTED BY OPERATOR'
STOP        : 03  000454  (2)3B04(2)
                          00286          STOP RUN.
                          00287
IF          : 03  000458  (2)3B04(3)
                          00288      IF WANT-INVOICES
DISPLAY     : 03  000474  (2)3B04(4)
                          00289          DISPLAY ' INVOICES not implemented'
MOVE        : 03  000498  (2)3B04(5)
                          00290          MOVE 0 TO OPTION-ENTRY (2).
                          00291
IF          : 03  0004AC  (2)3B04(6)
                          00292      IF WANT-STATEMENTS
DISPLAY     : 03  0004CC  (2)3B04(7)
                          00293          DISPLAY 'Enter statement message or CR'
ACCEPT      : 03  0004E4  (2)3B04(8)
                          00294          ACCEPT STANDARD-MESSAGE.
                          00295
OPEN        : 03  000514  (2)3B05(1)
                          00296
                          00297      OPEN INPUT CUSTOMER-FILE.
MOVE        : 03  000548  (2)3B05(2)
                          00298      MOVE 'NNNN999' TO CUST-CUST-NUMBER.
START       : 03  000560  (2)3B05(3)
                          00299      START CUSTOMER-FILE
                          00300          KEY IS > CUST-CUST-NUMBER.
                          00301
OPEN        : 03  000588  (2)3B05(4)
                          00302      OPEN OUTPUT STATEMENT-REPORT.
                          00303
                          00304  *****************************************************
                          00305
                          00306  MAIN-LINE SECTION.
                          00307  BEGIN.
READ        : 04  000000  (4)2B02(1)
                          00308
                          00309          READ CUSTOMER-FILE NEXT
                          00310              AT END
```

```
GO        | 28  2888|J  (0000225C)
                          00222                GO TO END=PROCESS.
ADD       F 68  2888|C  [0000226A)
                          00223            ADD 1 TO RECORD=COUNT.
                          00224        *
                          00225        *           Print statement if end-of-file.
                          00226        *
IF        I 28  288828  [0000227A)
                          00227            IF CUST-ONE=AMT > 2
PERFORM   I 38  288838  [0000228A)
                          00228                PERFORM PRINT-STATEMENT
ADD       I 28  288848  [0000229A)
                          00229                ADD 1 TO STATEMENT-COUNT.
                          00230        *
                          00231        *           If we need a mailing label for
                          00232        *           a catalog, print it.
                          00233        *
IF        I 28  288854  [000022BC)
                          00234            IF WANT-ALL-CATALOGS
                          00235                OR
                          00236                WANT-SOME-CATALOGS
                          00237                    AND
                          00238                CUST-BROUGHT-OUT = YES-CATALOG-FLAG
CALL      I 3A  288A44  [000022DA)
                          00239                    CALL "OCCNTR" USING CUST-CNT-FILE-RECORD
ADD       I 84  288A8C  [000022EA)
                          00240                    ADD 1 TO CATALOG-COUNT.
                          00241        *
                          00242        *           Check for discrepancies in the
                          00243        *           customer's records.
                          00244        *
MOVE      I 84  288AC8  [000022FA)
                          00245            MOVE ALL ZERO TO EXCEPTION-PIC-CODE.
IF        I 84  288AD4  [0000230A)
                          00246            IF CUST-CUSTOMER-NAME = SPACES
MOVE      I 84  288AE4  [0000231A)
                          00247                MOVE 1 TO EXCEPTION-INDICATOR (1).
IF        I 80  288B20  [0000232A)
                          00248            IF CUST-ADDRESS-LINE-1 = SPACES
                          00249                OR CUST-ADDRESS-ZIP-CODE NOT = "00000"
MOVE      I 84  288B78  [0000233A)
                          00250                    MOVE 1 TO EXCEPTION-INDICATOR (2).
IF        I 80  288B3C  [0000234A)
                          00251            IF CUST-PHONE = SPACES
MOVE      I 88  288B8C  [0000235A)
                          00252                MOVE 1 TO EXCEPTION-INDICATOR (3).
IF        I 84  288B98  [0000236A)
                          00253            IF CUST-CREDIT-LIMIT NOT = 0
MOVE      I 8C  288B78  [0000237A)
                          00254                MOVE 1 TO EXCEPTION-INDICATOR (4).
IF        I 8A  288B94  [0000238A)
                          00255            IF CUST-ONE-AMT > CUST-CREDIT-LIMIT
MOVE      I 8C  288BA4  [0000239A)
                          00256                MOVE 1 TO EXCEPTION-INDICATOR (5)
ELSE      I 8A  288BC8  [000023AA)
                          00257            ELSE
IF        I 84  288BC8  [000023BA)
                          00258            IF CUST-ONE-AMT + CUST-CREDIT-LIMIT > 0.8
MOVE      I 8C  288BE8  [000023CA)
                          00259                MOVE 1 TO EXCEPTION-INDICATOR (6).
IF        I 84  288C04  [000023DA)
                          00260            IF EXCEPTION-INDICATORS NOT = ALL ZERO
CALL      I 8A  288C14  [000023EA)
                          00261                CALL "EXCEPT" USING CUST-CNT-FILE-RECORD
                          00262                    EXCEPTION-INDICATORS.
                          00263        *
                          00264        *           Generate a "credit limit letter"
                          00265        *           if the customer has exceeded or
                          00266        *           is about to exceed his limit.
                          00267        *
IF        I A8  2C2234  [000024C0)
                          00268            IF WANT-CREDIT-LIMIT-LETTERS
                          00269                AND
                          00270            CUST-CREDIT-LIMIT NOT = CUST-CREDIT-LIMIT * 0.8
GO        I 84  2C2274  [000024D0)
                          00271                GO TO 90-EP.
                          00272        *
                          00273        *           Go get the next record.
                          00274        *
GO        I 88  2C227C  [000024E0)
                          00275            GO TO MAIN-LOOP.
                          00276
                          00277        90-EP.
CALL      I 38  2C82AC  [000024F0)
                          00278            CALL "CREDLTR" USING CUST-CNT-FILE-RECORD.
ADD       I 28  2C82A4  [0000250C)
                          00279            ADD 1 TO CREDIT-LIMIT-COUNT.
```

```
GO          S 24  0000287  (3000000093)
                             00346          GO TO PRINT-LINE.
                             00347
                             00348      ***************************************************
                             00349      *
                             00350      *               The CUSTOMER-FILE has been completely
                             00351      *               processed. Report significant counts.
                             00352      *
                             00353
                             00354       END-PROCESS SECTION 47.
                             00355       BEGIN.
CLOSE       S 25  0000202  (3000000082)
                             00356          CLOSE CUSTOMER-FILE.
CLOSE       S 25  0000230  (3000000C83)
                             00357          CLOSE STATEMENT-REPORT.
MOVE        S 25  0000215  (3000000C02)
                             00358          MOVE 'RECORD COUNTS TO DISP-MESSAGE.
MOVE        I 25  0002324  (3000000203)
                             00359          MOVE RECORD-COUNT TO DISP-NUM.
DISPLAY     I 25  0003218  (3000000183)
                             00360          DISPLAY DISP-MESSAGE.
MOVE        I 25  0003258  (3000000286)
                             00361          MOVE 'STATEMENTS' TO DISP-MESSAGE.
MOVE        I 25  0003305  (3000000314)
                             00362          MOVE STATEMENT-COUNT TO DISP-NUM.
DISPLAY     I 25  0003378  (3000000284)
                             00363          DISPLAY DISP-MESSAGE.
MOVE        I 25  0003440  (3000000400)
                             00364          MOVE 'INVOICES' TO DISP-MESSAGE.
MOVE        I 25  0003494  (3000000401)
                             00365          MOVE INVOICE-COUNT TO DISP-NUM.
DISPLAY     I 25  0003528  (3000000481)
                             00366          DISPLAY DISP-MESSAGE.
MOVE        I 25  0003578  (3000000478)
                             00367          MOVE 'CATALOGS' TO DISP-MESSAGE.
MOVE        I 25  0003620  (3000000501)
                             00368          MOVE CATALOG-COUNT TO DISP-NUM.
DISPLAY     I 25  0003664  (3000000581)
                             00369          DISPLAY DISP-MESSAGE.
MOVE        I 25  0003708  (3000000681)
                             00370          MOVE 'CREDIT LIMIT LETTERS' TO DISP-MESSAGE.
MOVE        I 25  0003760  (3000000781)
                             00371          MOVE CREDIT-LIMIT-COUNT TO DISP-NUM.
DISPLAY     I 25  0004004  (3000000881)
                             00372          DISPLAY DISP-MESSAGE.
STOP        I 25  0004050  (3000000981)
                             00373          STOP RUN.
                             00374
                             00375      ***************************************************
                             00376      *
                             00377      *               This section generates a statement
                             00378      *               for the current CUSTOMER-FILE
                             00379      *               record.
                             00380      *
                             00381
                             00382       PRINT-STATEMENT SECTION 48.
                             00383       BEGIN.
MOVE        I 26  0002300  (3000000A00)
                             00384          MOVE SPACES TO STATEMENT-REPORT-RECORD.
MOVE        I 26  0002340  (3000000A71)
                             00385          MOVE 'STATEMENT' TO FORM-NAME.
WRITE       I 26  0002368  (3000000B11)
                             00386          WRITE STATEMENT-REPORT-RECORD AFTER ADVANCING PAGE.
MOVE        I 26  0002398  (3000000C01)
                             00387          MOVE SPACES TO STATEMENT-REPORT-RECORD.
MOVE        I 26  0002430  (3000000D00)
                             00388          MOVE CUST-CUSTOMER-NAME TO ADDRESS-WINDOW.
MOVE        I 26  0002478  (3000000E00)
                             00389          MOVE 'DATE:' TO FORM-NAME.
MOVE        I 26  0002508  (3000000F00)
                             00390          MOVE TODAYS-REPORT-DATE TO FORM-DATE.
WRITE       I 26  0002538  (3000001000)
                             00391          WRITE STATEMENT-REPORT-RECORD AFTER ADVANCING 1 LINE.
MOVE        I 27  0002578  (3000001100)
                             00392          MOVE CUST-ADDRESS-LINE-1 TO ADDRESS-WINDOW.
MOVE        I 27  0002600  (3000001200)
                             00393          MOVE 'ACCT:' TO FORM-NAME.
MOVE        I 27  0002640  (3000001300)
                             00394          MOVE CUST-CUST-NUMBER TO FORM-DATE.
WRITE       I 27  0002668  (3000001400)
                             00395          WRITE STATEMENT-REPORT-RECORD AFTER ADVANCING 1 LINE.
MOVE        I 28  0002700  (3000001500)
                             00396          MOVE SPACES TO STATEMENT-REPORT-RECORD.
MOVE        I 28  0002738  (3000001600)
                             00397          MOVE CUST-ADDRESS-LINE-2 TO ADDRESS-WINDOW.
WRITE       I 28  0002768  (3000001700)
                             00398          WRITE STATEMENT-REPORT-RECORD AFTER ADVANCING 1 LINE.
MOVE        I 28  0003000  (3000001800)
                             00399          MOVE CUST-ADDRESS-LINE-3 TO ADDRESS-WINDOW.
```

```
MOVE      1 Ab  ##2##C  (0H@A9A@@()
                        @@4@@
WRITE     1 Aa  ##0L2#  (@HH@HC@*C)
                        @H#@L
MOVE      1 @@  ##0L2#  (@@**H@24)
                        @A@A2
MOVE      1 Ab  ##0L2)  (%3@@HH%A)
                        @@4@3
MOVE      1 Ab  @#0L2#  (=4@*@@@@4)
                        @#u@4
WRITE     1 3A  @@.L5*  (%4*H@@5C)
                        @Hu@5
MOVE      1 Aa  @@#L2#  (3A*H#B7A)
                        %0J@6
MOVE      1 Ab  @#0LA)  (%3@*H@@3)
                        @F%#T
MOVE      1 Aa  *@@L4C  (%*@*#H*2)
                        @H#H@
MOVE      1 @6  @@#L#)  (@A#@@A%J)
                        3@u@9
WRITE     1 @6  @#@1#C  (%*#H#H@3)
                        @H#J@
MOVE      1 @6  @#@LC#  (%4H@#BCC)
                        @@#11
IF        1 Aa  #H#1C3  (%3@*H@@H)
                        A9#12
MOVE      1 4b  @@#1E3  (%%@3#BC#)
                        %%413
                        @@L1#
ELSE      1 %b  #H#1F)  (%3@@@F#)
                        ;%%15
IF        1 @6  @@81F@  (?3%o%BFC)
                        2%%16
MOVE      1 %*  @@#2)#  (2@2o%E1C)
                        ;7%17
                        %%18
ELSE      1 *%  @#@22)  (/A%o4C20)
                        @%%9
MOVE      1 %%  #H#2#C  (?2?3@C3#)
                        @?%2@
WRITE     1 2*  3A@230  (*@?3%;3C)
                        @?%21
                        ;7%22
                        @?%3
EXIT      1 @@  3?@25C  (%@@*@C%#)
                        #V%2%
```

```
    MOVE CUST-ADDRESS-ZIP-CODE TO ADDRESS-ZIP.

    WRITE STATEMENT-REPORT-RECORD AFTER ADVANCING 1 LINE.

    MOVE SPACES TO STATEMENT-REPORT-RECORD.

    MOVE CUST-CREDIT-LIMIT TO REPORT-CREDIT.

    MOVE CUST-AMOUNT TO REPORT-AMT.

    WRITE STATEMENT-REPORT-RECORD AFTER ADVANCING M LINES.

    MOVE SPACES TO STATEMENT-REPORT-RECORD.

    MOVE TODAYS-REPORT-DATE TO STATEMENT-DATE.

    MOVE CUST-NAME-ADDR TO STATEMENT-BALANCE.

    MOVE "**,###.## DUE" TO STATEMENT-CAPTION.

    WRITE STATEMENT-REPORT-RECORD AFTER ADVANCING 6 LINES.

    MOVE SPACES TO STATEMENT-REPORT-RECORD.

    IF CUST-AMT > CUST-CREDIT-LIMIT

        MOVE "*** CREDIT LIMIT EXCEEDED ***"
            TO STATEMENT-REPORT-RECORD

    ELSE

    IF CUST-AMT > CUST-CREDIT-LIMIT * 0.8

        MOVE "CONSIDER AN INCREASED CREDIT LIMIT, "
            TO STATEMENT-REPORT-RECORD

    ELSE

        MOVE STANDARD-MESSAGE TO STATEMENT-REPORT-RECORD.

    WRITE STATEMENT-REPORT-RECORD AFTER ADVANCING 4 LINES.

END".

    EXIT.
```


STATB
COBVFB R12B0A
DATA MAP
LEVEL
NAME
SOURCE LINE
BGIN LOCN
DIM LOC
USAGE
CLASS
OCC
LENGTH
FD CUSTOMER-FILE
FC STATEMENT-REPORT
01 CUSTOMER-FILE-RECORD
03 CUST-CUST-NUMBER
03 CUST-CUSTOMER-NAME
03 CUST-ADDRESS-LINE-1
03 CUST-ADDRESS-LINE-2
03 CUST-ADDRESS-LINE-3
03 CUST-ADDRESS-ZIP-CODE
03 CUST-PHONE
05 CUST-PHONE-AREA-CODE
05 CUST-PHONE-EXCHANGE
05 CUST-PHONE-LAST-4
03 CUST-PHONE-NUMBER
03 CUST-ATTENTION-LINE
03 CUST-CREDIT-LIMIT
03 CUST-HEADER-DATE
05 NEXT-ACCT-NUMBER
03 CUST-ONE-AMT
03 CUST-BOUGHT
03 CUST-NEXT-ORDER-SEQUENCE
03 CUST-NEXT-PAYMENT-SEQUENCE
01 STATEMENT-REPORT-RECORD
03 ADDRESS-NEXCOM
03 ADDRESS-ZIP
03 FORM-NAME
05 FORM-DATE
01 S-P-R-Z
03 REPORT-CREDIT
03 REPORT-YTD
01 S-C-R-Z
03 STATEMENT-DATE
03 STATEMENT-CAPTION
03 STATEMENT-BALANCE
01 CUSTOMER-FILE-STATUS
01 STATEMENT-REPORT-STATUS
01 CUSTOMER-FILE-ID
01 TODAYB-DATE
01 IDR
03 TODAY-YEAR
03 TODAY-MONTH
03 TODAY-DAY
01 TODAYB-REPORT-DATE
03 TODAY-MONTH
03 TODAY-DAY
03 TODAY-YEAR
01 STANDARD-MESSAGE
01 DISP-MESSAGE
03 DISP-NUM
01 IPD-DATALOG-PIN-NUM
01 EXCEPTION-PROCEDURES
03 EXCEPTION-PROCEDURE
01 OPTION8-AREA
03 OPTION8-AREA-CHAR
01 X-COUNT
01 OPTION4-STORAGE
03 OPTION4-ENTRY
01 OPTION4-VALUES
01 RECORD-COUNT
01 STATEMENT-COUNT
01 INVOICE-COUNT
01 CREDIT-LIMIT-COUNT
01 CAT-LOG-COUNT
DISP
DISP
DISP

```
STATB                                          12-Jan-1979 08:40:15    VAX-11 COBOL-74  V4.00-01
IDENT: 012286                                                          $T4TB,COB;4

                                (29)
                        PROCEDURE NAME MAP

NAME                    SOURCE      PSECT       OFFSET  SEG   SECT  PARA
(30)                    LINE        (32)        (33)    (34)  (35)  (36)
                        (31)
CUBTCH=ERROR            00182   $1ATB$$$$$$$$01 00000C0  00    0     P
SBEGIN                  00183   $14TB$$$$$$$$01 00000C0  00          P
STATE=ERROR             00194   $TATB$$$$$$$$02 0000000  00    0
SBEGIN                  00198   $TATB$$$$$$$$02 0000000  00          P
START-UP=HOUSEKEEPING   00205   $14TB$$$$$$$$01 0000000  00    0
SBEGIN                  00206   $14TB$$$$$$$$01 0000000  00          P
ENDFILE=OPTION0         00254   $14TB$$$$$$$$03 0000340  00          P
MAINLINE                00283   $TATB$$$$$$$$0A 0000000  00    0
SBEGIN                  00285   $14TB$$$$$$$$0A 0000000  00          P
DO-CR                   00343   $14TB$$$$$$$$04 00002BC  00          P
END-PROCESS             00354   $TATB$$$$$$$$05 0000000  00    0
SBEGIN                  00355   $14TB$$$$$$$$05 0000000  00          P
PRINT-STATEMENT         00382   $TATB$$$$$$$$06 0000000  00    0
SBEGIN                  00383   $14TB$$$$$$$$06 0000000  00          P
SEXIT                   00423   $14TB$$$$$$$$06 00002BC  00          P
```

```
STATB                                          12-Jan-1979 08:40:15    VAX-11 COBOL-74  V4.00-01
IDENT: 012286                                                          BTATB,COB;4

                                (37)
                        SEGMENTATION MAP

   (38)      SECTION NAME      SEGMENT NO.      NAME (40)      (41) SIZE (42)
                                   (39)
CUBTCH=ERROR                    00          $14TB$$$$$$$$01    00000C0     000C8
STATE=ERROR                     00          $TATB$$$$$$$$02    0000000     00000
START-UP=HOUSEKEEPING           00          $14TB$$$$$$$$03    0000340     01356
MAINLINE                        00          $TATB$$$$$$$$04    00002BC     00128
END-PROCESS                     00          $TATB$$$$$$$$05    0000000     00332
PRINT-STATEMENT                 00          $TATB$$$$$$$$06    0000000     00624
```

```
STATB                                          12-Jan-1979 08:40:15    VAX-11 COBOL-74  V4.00-01
IDENT: 012284                                                          BTATB,COB;4

                        (43)
COMPILER GENERATED PSECTS

NAME (44)               (45) SIZE (46)
BTATB$$$$000000         000024     00036
```

```
BTATB                                                   12-Jan-1979 08:30:15    VAX-11 CORDL-T0  V3.00-01
IDENT; B12000                                           STATB.CO$p1

                    (47)
REFERENCED DTB ROUTINES

C7%$HDME    C7488*OPEN    C7480XMBGV    C7%5XMMIT    C7459*STAR    C7459*CLOS
C7%$KEACC   C7489*E?IE    C7480XGO      C7%63EHDP    C7459*ST*K    C7459*JNBM
C7%$KJNTB   C7A59]*?*D    C7480X4CC8    C7%6YJMIT    C7A4*[2DLT    C7459[52LT
C7%5C6PLT   C7A5C2>LE     C7A5CGP%E     C7%5C$1EN    C7A5[*5ED     C7A5C61ME
C7%5EF$ME   C7A5C1?G1     C7480ZGGF     C7%5C6PET    C7A59*D4D     C7A5*]I5L
C7%$MDPBT   C7A594546      C7480*OL*     C7%5AAF20    C7A5**M3>P    C7A5A4]*1
C7%$K*RF]
```

```
BTATB                                                   12-Jan-1979 08:40:15    VAX-11 C0B0L-T0  V3.00-01
IDENT; B12000                                           BTATB.CO$p1

DATA PSECT MAP (48)

NAME (49)          (50)  SIZE (51)
BTATB$$$$$$$DAT    ?00000    01F3C
BTATB$$$$$$$DPD    000380    00F00
BTATB$$$$$$$ARG    ?3?000    00000
BTATB$$$$$$$MRK    ?2?01A    00026
BTATB$$$$$$$LIT    000250    00?04
BTATB$$$$$$$LTD    000250    00?20
BTATB$$$$$$$DT     000200    00000
BTATB$$$$$$$USE    000?2?    00052
SCBXAL             000200    00008
SCBFOL             000?2?    00004
```

```
BTATB                                                   12-Jan-1979 08:40:15    VAX-11 C0B0L-T0  V3.00-01
IDENT; B12000                                           BTATB.CO$p1

EXTERNAL SUBPROGRAM REFERENCES (52)

DOC416    EXCEP    CREULP
```

```
BTATB                                                   12-Jan-1979 08:40:15    VAX-11 C0B0L-T0  V3.00-01
(53) IDENT; B12000 (54)                                 BTATB.CO$p1

SEVERITY    ERROR COUNT
    )           )
```

```
DUE418                                                  32=Jan=1979 08:41:22    *A2=|| COBOL=74,02=03
IDENT; 012945                                                                   DUE418,COBOL
/*JOB(ACCT,PGM=BF       /COPY  151          /*USER=686_REFERENCE  /*SET=DUE3T8
/*JOB                  /*CONFIGURATION      /*####|x00           /*SET=DUEDATA
/*SET/SW=1=ACE=NO=
                    000001        IDENTIFICATION DIVISION.
                    000002        PROGRAM-ID.
                    000003            DUE418.
                    000004        DATE-WRITTEN.    * 24 JANUARY 1979.
                    000005        DATE-COMPILED.
                    000006
                    000007        *            This subprogram prints a mailing label
                    000008        *         for each CUSTOMER-FILE record passed from
                    000009        *         the calling program.
                    000010        *
                    000011
                    000012            32=Jan=1979 ,
                    000013        ENVIRONMENT DIVISION.
                    000014
                    000015        CONFIGURATION SECTION.
                    000016        SOURCE-COMPUTER.        ###=||,
                    000017        OBJECT-COMPUTER.        ###=||,
                    000018
                    000019        INPUT-OUTPUT SECTION.
                    000020        FILE-CONTROL.
                    000021            SELECT LABEL-REPORT
                    000022                ASSIGN TO PLANEL*
                    000023                FILE STATUS IS LABEL-REPORT-STATUS.
                    000024
                    000025        DATA DIVISION.
                    000026
                    000027        FILE SECTION.
                    000028
                    000029        FD  LABEL-REPORT
                    000030            LABEL RECORDS ARE STANDARD.
                    000031        01  LABEL-REPORT-RECORD          PIC X(30).
                    000032        01  WS-DETAIL.
                    000033            05  FILLER                   PIC      X(14).
                    000034            05  WS-COUNT                 PIC      9(6).
                    000035        01  WS-DETAIL-2.
                    000036            05  FILLER                   PIC      X(32).
                    000037            05  WS-ZIP                   PIC      9(5).
                    000038
                    000039        WORKING-STORAGE SECTION.
                    000040
                    000041        01  LABEL-REPORT-STATUS          PIC 9(2)
                    000042            VALUE "##".
                    000043
                    000044        LINKAGE SECTION.
                    000045
                  L 000046        COPY CUSTFD.
                  L 000047
                  L 000048
                  L 000049        01  CUSTOMER-FILE-RECORD.
                  L 000050            05  CUST-CUSTOMER-NUMBER                 PIC      X(6).
                  L 000051            05  CUST-CUSTOMER-NAME                   PIC      X(30).
                  L 000052            05  CUST-ADDRESS-LINE-1                  PIC      X(30).
                  L 000053            05  CUST-ADDRESS-LINE-2                  PIC      X(30).
                  L 000054            05  CUST-ADDRESS-LINE-3                  PIC      X(30).
                  L 000055            05  CUST-ADDRESS-ZIP-CODE                PIC      9(5).
                  L 000056            05  CUST-PHONE-GROUP.
                  L 000057                10  CUST-PHONE-AREA-CODE             PIC      X(3).
                  L 000058                10  CUST-PHONE-EXCHANGE              PIC      X(3).
                  L 000059                10  CUST-PHONE-LAST-4                PIC      9(4).
                  L 000060            05  CUST-PHONE-NUMBER
                  L 000061                REDEFINES CUST-PHONE                  PIC      9(10).
                  L 000062            05  CUST-ATTENTION-LINE                  PIC      X(24).
                  L 000063            05  CUST-CREDIT-LIMIT                    PIC      9(3)V99.
                  L 000064            05  CUST-HEADER-DATA REDEFINES CUST-CREDIT-LIMIT.
                  L 000065                10  FILLER                           PIC      X(6).
                  L 000066                10  NEXT-ACCT-NUMBER                 PIC      9(6).
                  L 000067            05  CURR-BALANCE                         PIC      9(3)V99.
                  L 000068            05  CURR-AMOUNT                          PIC      9(3)V99.
                  L 000069            05  CUST-LAST-ORDER-SEQUENCE             PIC      9(4).
                  L 000070            05  CUST-LAST-PAYMENT-SEQUENCE           PIC      9(4).
                  L 000071
                    000072        PROCEDURE DIVISION USING
                    000073            CUSTOMER-FILE-RECORD.
                    000074
                    000075        DECLARATIVES.
                    000076
                    000077        REPORT-ERROR SECTION.
LBL       I USE   000078  (0020000)
                    000079            USE AFTER STANDARD ERROR PROCEDURE ON LABEL-REPORT.
                    000080        DECLARATIVES.
```

```
DISPLAY    1 01  0400B7  (2)0034(28)
                          22052          DISPLAY "END STATUS OF LAST REPORT: CODE ="
                          22053              LABEL=REPORT-STATUS
                          22054              "!".
                          22055
                          22056      END DECLARATIVES.
                          22057
                          22058      PROCEDURE SECTION.
                          22059      BEGIN.
IF         2 02  000000  L(0222)(2)(3)
                          22061
                          22062          IF LABEL=REPORT-STATUS = "XXX"
OPEN       2 02  000010  (0)(0)(4)(2)(5)
                          22064              OPEN OUTPUT LABEL=REPORT,
MOVE       2 02  000016  (00)(0)(0)(2)(6)
                          22066          MOVE SPACES TO LABEL-REPORT-RECORD,
MOVE       2 02  000026  (0)(0)(0)(0)(6)(3)
                          22068          MOVE CUST-CUSTOMER-NUMBER TO LP-ACCOUNT,
WRITE      2 02  000034  (0)(0)(0)(0)(4)(3)
                          22070          WRITE LABEL=REPORT-RECORD
                          22071              AFTER ADVANCING 1 LINE,
MOVE       2 02  000050  (0)(0)(0)(0)(0)(0)
                          22073          MOVE CUST-CUSTOMER-NAME TO LABEL-REPORT-RECORD,
WRITE      2 02  000056  (0)(0)(0)(0)(0)(0)
                          22075          WRITE LABEL-REPORT-RECORD
                          22076              AFTER ADVANCING 2 LINES,
MOVE       2 02  000072  (0)(0)(0)(0)(0)(0)
                          22078          MOVE CUST-ADDRESS-LINE-1 TO LABEL-REPORT-RECORD,
WRITE      2 02  000100  (0)(0)(0)(0)(0)(0)
                          22080          WRITE LABEL-REPORT-RECORD
                          22081              AFTER ADVANCING 1 LINE,
MOVE       2 02  000116  (0)(0)(0)(0)(0)(0)
                          22083          MOVE CUST-ADDRESS-LINE-2 TO LABEL-REPORT-RECORD,
WRITE      2 02  000124  (0)(0)(0)(0)(0)(0)
                          22085          WRITE LABEL=REPORT-RECORD
                          22086              AFTER ADVANCING 1 LINE,
MOVE       2 02  000142  (0)(0)(0)(0)(0)(0)
                          22088          MOVE CUST-ADDRESS-LINE-3 TO LABEL-REPORT-RECORD,
MOVE       2 02  000150  (0)(0)(0)(0)(0)(0)
                          22090          MOVE CUST-ADDRESS-ZIP-CODE TO LP-ZIP,
WRITE      2 02  000156  (0)(0)(0)(0)(0)(0)
                          22092          WRITE LABEL=REPORT-RECORD
                          22093              AFTER ADVANCING 1 LINE,
MOVE       2 02  000174  (0)(0)(0)(0)(0)(0)
                          22095          MOVE SPACES TO LABEL-REPORT-RECORD,
WRITE      2 02  000202  (0)(0)(0)(0)(0)(0)
                          22097          WRITE LABEL-REPORT-RECORD
                          22098              AFTER ADVANCING 2 LINES.
EXIT       2 02  000220  (0)(0)(0)(0)(0)(0)
                          22100          EXIT PROGRAM.
```

```
DOCATA                                                    32-JAN-1979 02:43:22    VAX-11 COBOL V3.0  Page  8
ZDENT1 012006                                                                     DCEXTA,COB;6

                                    Data Map

LEVEL          NAME                 $OWNLE ODEV   SIZE   USAGE CLASS DEC LENGTH
                                     LINE  LOCN    LOC

FD    LABEL=REPORT                  22029 00000000
01    LABEL=REPORT-RECORD           22031 000140 022220 DISP  AN      20  00000
01    L=R=DETAIL                    22032 000140 000000 DISP  AN      20  00000
  05  LP-ACCOUNT                    22034 000140 000000 DISP  AN      07  00000
01    L=R=DETAIL=2                  22035 000140 000000 DISP  AN      09  00037
  05  LP-ZIP                        22037 000140 000010 DISP  AN      09  00005
01    LABEL=REPORT=STATUS           00041 000178 000020 DISP  AN      02  00002
LL    CUSTOMER=FILE=RECORD          00049 000000 000000 ISAM  AN      80  00200
L 05  CUST=CUST=NUMBER              00050 000000 000000 ISAM  AN      06  00006
L 05  CUST=CUSTOMER=NAME            00051 000006 000000 DISP  AN      20  00020
L 03  CUST=ADDRESS=LINE=1           00052 000026 000000 DISP  AN      20  00020
L 03  CUST=ADDRESS=LINE=2           00053 000046 000020 DISP  AN      20  00020
L 03  CUST=ADDRESS=LINE=3           00054 000066 000040 DISP  AN      20  00020
L 03  CUST=ADDRESS=ZIP=CODE         00055 000086 000060 DISP  AN      09  00009
L 03  CUST=PHONE                    00056 000115 ******* DISP  AN      08  00008
L 05  CUST=PHONE=AREA=CODE          00057 000115 ******* DISP  AN      03  00003
L 05  CUST=PHONE=EXCHANGE           00058 000120 ******* DISP  AN      03  00003
L 05  CUST=PHONE=LAST=4             00059 000123 ******* DISP  AN      04  00004
L 03  CUST=PHONE=NUMBER             00060 000115 ******* DISP  AN      07  00007
L 03  CUST=ATTENTION=LINE           00062 000000 ******* DISP  AN      20  00020
L 03  CUST=CREDIT=LIMIT             00063 000127 ******* DISP  NUM     08  00012
L 03  CUST=RESERVED=DATA            00064 000000 ******* DISP  AN      20  00020
L 05  NEXT=ACCT=NUMBER              00066 000000 ******* DISP  NUM     03  00003
L 03  CUST=CODE=ANT                 00067 000000 ******* DISP  NUM     02  00002
L 03  CUST=CODE=P                   00068 000000 ******* DISP  NUM     02  00002
L 03  CUST=RECORD=SEQUENCE          00071 000000 ******* DISP  NUM     02  00002
L 03  CUST=NEXT=RECORD=SEQUENCE     00072 000000 ******* DISP  NUM     04  00004
```

```
DOCAT8                                                  12-Jan-1979 08:41:22    VAX-11 COBOL-74  V4.00-81
IDENT: 812286                                           DOCAT8.COB;6


                        PROCEDURE NAME MAP

NAME                    SOURCE      PSECT       OFFSET SEG  SECT  PARA
                        LINE

REPORT-ERROR             00870  DOCAT$$$$$$$$0001  000230  00     0
SBEGTH                   00881  DOCAT$$$$$$$00M1   000000  00         P
MAINLINE                 00900  DOCAT$$$$$$$00M2   000000  00     0
SBEGIH                   00909  DOCAT$$$$$$$$0002  000260  00         P


DOCAT8                                                  12-Jan-1979 08:41:22    VAX-11 COBOL-74  V4.00-81
IDENT: 812286                                           DOCAT8.COB;6


                        SEGMENTATION MAP

          SECTION NAME        SEGMENT NO.        NAME              SIZE

 REPORT-ERROR                    00        DOCAT$$$$$$$00M1    000230    00056
 MAINLINE                        00        DOCAT$$$$$$$00M2    000310    00316


DOCAT8                                                  12-Jan-1979 08:41:22    VAX-11 COBOL-74  V4.00-81
IDENT: 812286                                           DOCAT8.COB;6

COMPILER GENERATED PSECTS

NAME                    SIZE

DOCAT$$$000000000    000020    00010
DOCAT$$$$000000003   000040    00064


DOCAT8                                                  12-Jan-1979 08:41:22    VAX-11 COBOL-74  V4.00-81
IDENT: 812286                                           DOCAT8.COB;6

REFERENCED CTB ROUTINES

CTB$OPEN    CTB$READ    CTB$XEQ15    CTB$XGO     CTB$XERROR    CTB$XEXIT
CTB$$BJBK   CTB$$INIT   CTB$CBYTE    CTB$PJL$L   CTB$RPCPBT    CTB$XHGLA


DOCAT8                                                  12-Jan-1979 08:41:22    VAX-11 COBOL-74  V4.00-81
IDENT: 812286                                           DOCAT8.COB;6

DATA PSECT MAP

NAME                    SIZE

DOCAT$$$$$$$04T     000178    00376
DOCAT$$$$$$$0DD     000030    00048
DOCAT$$$$$$$499     000044    00072
DOCAT$$$$$$$VRK     000010    00026
DOCAT$$$$$$$L31     000028    00040
DOCAT$$$$$$$L7D     000038    00060
DOCAT$$$$$$$4P1     000000    00000
DOCAT$$$$$$$U5E     000020    00032
$CBXA1              000004    00004
SCBPOL              000004    00004


DOCAT8                                                  12-Jan-1979 08:41:22    VAX-11 COBOL-74  V4.00-81
IDENT: 812286                                           DOCAT8.COB;6

EXTERNAL SUBPROGRAM REFERENCES

NO EXTERNAL SUBPROGRAM REFERENCES

NO ERRORS
```

APPENDIX D

DIAGNOSTIC ERROR MESSAGES

This Appendix contains a numerical listing of the diagnostic messages generated by the compiler. Following the text of most messages are explanations of the diagnostics, including descriptions of the compiler's recovery actions.

001 CONTINUE PUNCH WITH BLANK STATEMENT. IGNORED.

A blank line has a continue indicator. The continue indicator is ignored.

002 QUOTE OR CONTINUE PUNCH MISSING. QUOTE ASSUMED.

A non-numeric literal has no quote and the following line has no continuation indicator. A terminal quote is assumed at the end of the line.

003 VIOLATION OF AREA A. ASSUMED CORRECT.

The first non-blank character on a continued line occurs in Area A. The error is ignored.

004 LINE LENGTH EXCEEDS INPUT BUFFER. TRUNCATED.

Continuation lines cause a COBOL word to exceed the capacity of the input buffer. The word is truncated on the right; the number of characters retained depends on the type of word being processed.

005 .IO CONTROL. WITHOUT .FILE CONTROL. IGNORED.

An I-O-CONTROL paragraph appears when no FILE-CONTROL paragraph was present. The I-O-CONTROL paragraph is ignored.

006 .STRING. DATA ITEM MUST HAVE DISPLAY USAGE.

A data item in a STRING statement is not defined with DISPLAY usage. Fatal.

007 NAME EXCEEDS 30 CHARACTERS. TRUNCATED TO 30.

A character-string that appears to be a name exceeds 30 characters in length. The string is truncated on the right to 30 characters.

010 NUMERIC LITERAL OVER 18 DIGITS. TRUNCATED TO 18.

A numeric literal exceeds 18 digits in length. The literal is truncated on the right, with any necessary adjustment to scaling. The sign is retained.

011 NUMERIC LITERAL HAS MULTIPLE DECIMAL POINTS.

A numeric literal has more than one decimal point.

012 PICTURE CLAUSE ILLEGAL ON GROUP LEVEL. IGNORED.

A group level item has a PICTURE clause. The clause is ignored.

013 .SELECT. NOT FOUND. SENTENCE IGNORED.

A FILE-CONTROL statement should begin with the word SELECT, but does not. All words up to the next period are ignored.

014 JUST.SYNC.BLANK CLAUSES WRONG AT GROUP. IGNORED.

A group level item may not contain JUSTIFIED, SYNCHRONIZED, or BLANK WHEN ZERO clauses. The clause is ignored.

015 FILENAME MISSING OR INVALID. SELECT IGNORED.

A SELECT statement either contains no user name or the the user name is invalid. The SELECT statement is ignored.

016 USAGE CONFLICTS WITH GROUP USAGE. USES GROUP.

The usage specified for this item differs from the usage stated at a higher group level. The group level usage is used.

017 ILLEGAL NUMERIC DATANAME IN .STRING.

A numeric data item in a STRING statement has an illegal description. Fatal.

020 .ALL. ILLEGAL IN CONTEXT OF .STRING. STATEMENT.

An ALL literal has been used in a STRING statement. Fatal.

021 SYNTAX ERROR OR NO TERMINATOR. CLAUSES SKIPPED.

A SELECT statement is missing its terminating period, or an error causes the statement to be processed before all clauses were found. The SELECT statement is ignored.

022 NUMERIC LITERAL ILLEGAL IN THIS STATEMENT.

A STRING, UNSTRING, or INSPECT statement contains a numeric literal. Fatal.

023 SENDING LIST OMITTED IN .STRING. STATEMENT.

A STRING statement contains no sending fields before a DELIMITED BY phrase. Fatal.

024 MORE THAN ONE FILENAME IN .ASSIGN.

The non-numeric literal of an ASSIGN clause contains more than one file specification. Only the first specification is used.

025 ILLEGAL DATANAME FOLLOWS .INTO. IN .STRING.

The receiving field of a STRING statement is invalid. Fatal.

026 SUBSCRIPTING DEPTH EXCEEDS 3. OVER 3 IGNORED.

The OCCURS clause is nested more than three deep. The clause is ignored.

027 VALUE ILLEGAL IN OCCURS ITEM. IGNORED.

A VALUE clause appears in an item with an OCCURS clause or in an item subordinate to an OCCURS clause. The VALUE clause is ignored.

030 VALUE ILLEGAL IN REDEFINES ITEM. IGNORED.

A VALUE clause appears in an item that either contains a REDEFINES clause or is subordinate to an item with a REDEFINES clause.

031 NO TERMINATOR FOR .IO CONTROL. PARAGRAPH.

The I-O-CONTROL paragraph is not terminated by a period. The terminator is assumed present.

032 .MAP. NO LONGER APPLICABLE. IGNORED.

An APPLY clause with the MAP option is not applicable for this version and future versions of the compiler. The APPLY clause is ignored.

033 AN IO CONTROL CLAUSE WITHOUT FILES.

A file-name is missing in a clause of the I-O-CONTROL paragraph. The clause is ignored.

034 SYNTAX ERROR IN .APPLY.

An APPLY clause has illegal syntax. The clause is ignored.

035 INVALID ACCESS MODE. TREAT AS SEQUENTIAL.

The SELECT statement contains an invalid ACCESS mode. SEQUENTIAL ACCESS mode is assumed.

036 INVALID FILE ORGANIZATION. TREAT AS SEQUENTIAL.

The SELECT statement contains an invalid ORGANIZATION specification. SEQUENTIAL organization is assumed.

037 NO SELECT STATEMENTS.

A FILE-CONTROL paragraph either contains no SELECT statements or none of those present is valid. The FILE-CONTROL paragraph is ignored.

040 .ASSIGN. OMITTED FROM SELECT. SELECT IGNORED

A SELECT statement contains no ASSIGN clause. The SELECT statement is ignored.

041 DECIMAL PLACES TRUNCATED.

Decimal places have been truncated from a numeric literal during conversion for use as an integer. The integer positions are used.

042 INTEGER EXPECTED, ZERO ASSUMED.

An integer literal was expected, but fractional positions were found. The literal is ignored and a value of zero is assumed.

043 INTEGER VALUE TOO BIG. LARGEST VALUE USED.

A numeric literal is too big for conversion as an integer in the given context. A value of 32,767 is used.

044 ERROR IN DATA RECORDS CLAUSE. CLAUSE SKIPPED.

The word DATA is not followed by RECORD or RECORDS in the DATA RECORDS clause. The DATA RECORDS clause is ignored.

045 ERROR IN LABEL RECORDS CLAUSE. CLAUSE SKIPPED.

The word LABEL is not followed by RECORD or RECORDS in the LABEL RECORDS clause. The LABEL RECORDS clause is ignored.

046 NO INTEGER IN BLOCK CLAUSE. CLAUSE SKIPPED.

The BLOCK clause does not contain a numeric literal. The BLOCK clause is ignored.

047 BAD VALUE IN BLOCK CLAUSE. CLAUSE SKIPPED.

The numeric literal in the BLOCK clause causes an illegal block size. The block size in bytes must be greater than 0 and less than 32768. Clause ignored.

050 NO INTEGER IN RECORD CLAUSE. CLAUSE SKIPPED.

The RECORD CONTAINS clause does not contain a numeric literal. The RECORD CONTAINS clause is ignored.

051 INVALID VALUE IN RECORD CLAUSE. CLAUSE SKIPPED.

The numeric literal in the RECORD CONTAINS clause is not greater than zero. The RECORD CONTAINS clause is ignored.

052 INVALID FILENAME. FD SKIPPED.

The word following FD is not valid as a file-name. The FD entry is ignored.

053 FD TERMINATOR MISSING. ASSUMED PRESENT.

The file description entry contains no period terminator. The error is ignored.

054 KEY WORD EXPECTED. REMAINING CLAUSES SKIPPED.

A keyword that begins a clause, such as BLOCK, LABEL, DATA, etc., is missing. The remainder of the FD entry is ignored.

055 NO LABEL CLAUSE IN FD. STANDARD ASSUMED.

The FD entry contains no LABEL RECORD clause. LABEL RECORD IS STANDARD is assumed.

056 NO SELECT. FILE DELETED.

The FD entry's file-name has no corresponding SELECT statement. The FD entry is ignored. All references to the file-name will be diagnosed as undefined.

057 ALLOCATED SPACE EXCEEDS LARGEST RECORD.

The maximum record size specified by the RECORD CONTAINS clause exceeds the space required for any 01 entry under the same file. The value specified by the RECORD CONTAINS clause is used.

060 RECORD AREA EXTENDED TO CONTAIN LARGEST RECORD.

The space required by the largest 01 record under a file description exceeds the space required by the RECORD CONTAINS clause in the FD entry. The value derived from the 01 record description is used.

061 NO RECORD AREA. FILE DELETED.

No record area is allocated for a file description. The file description is ignored. All references to the file will be diagnosed as undefined.

062 ILLEGAL DATANAME FOLLOWS .WITH POINTER. PHRASE.

The data item used as a pointer in a STRING or UNSTRING statement is illegal. Fatal.

063 ILLEGAL SYNTAX IN .STRING. STATEMENT.

A STRING statement contains illegal syntax. Fatal.

064 77 ILLEGAL IN FILESECTION. CHANGED TO 01.

A 77 level item description has been found in the FILE SECTION. The 77 level is treated as an 01 level.

065 ILLEGAL WORD FOLLOWS .DELIMITED BY. PHRASE.

A data-name or literal is expected following a DELIMITED BY phrase in a STRING or UNSTRING statement. Fatal.

066 ILLEGAL USE OF .ALL.. IGNORED.

In the VALUE clause, an ALL numeric literal is detected. ALL is ignored by the compiler.

067 CONDITION NAME MISSING OR INVALID. 88 IGNORED.

The condition-name in an 88 level entry is either missing or invalid. The entire entry is ignored.

070 TWO INDEXED KEYS START AT SAME OFFSET IN RECORD.

The leftmost character position of the RECORD KEY or ALTERNATE RECORD KEY data-name corresponds to the leftmost character position of some other RECORD KEY or ALTERNATE RECORD KEY data-name. The clause is ignored.

071 .REDEFINES. ON 01 LEVEL IN FILE SECTION INVALID.

The REDEFINES clause is present on the 01 level in the FILE SECTION, where redefinition is implicit. REDEFINES clause is ignored.

072 PICTURE IGNORED FOR INDEX ITEM.

An item defined as USAGE INDEX has a PICTURE clause. The PICTURE clause is ignored.

073 NONNUMERIC PIC ON COMP ITEM. TREATED AS DISPLAY.

An item defined with non-DISPLAY usage has a picture-string with non-numeric characters. The stated usage is ignored. The item is treated as USAGE DISPLAY.

074 SUBSCRIPT OUT OF RANGE. ASSUME 1.

A literal subscript is either less than 1 or greater than the maximum allowable value. A value of 1 is used.

075 .STATUS. OMITTED FROM .FILE STATUS.. ASSUMED.

The FILE STATUS clause has incorrect syntax. The error is ignored.

076 SOME FILES WITHOUT POSIT. NO. IN MUL. FILE TAPE.

A MULTIPLE FILE TAPE clause contains file-names with POSITION clauses. Not all the file-names contain POSITION clauses. The error is ignored. File searching during OPEN will find the file.

077 .MULTIPLE FILE TAPE. SYNTAX ERROR.

A MULTIPLE FILE TAPE clause contains a syntax error. The clause is ignored.

100 OPERAND CLASSES IN CONFLICT.

One or more operands in a statement have an invalid class. Fatal.

101 POSSIBLE RECEIVING FIELD TRUNCATION.

A MOVE statement results in right-hand truncation of the receiving field value. This is not an error and is ignored.

102 TOO FEW SOURCE FIELDS FOR ADD .GIVING..

At least two valid source operands must appear in an ADD...GIVING statement. Fatal.

103 .EXIT. WAS NOT THE ONLY VERB IN PARAGRAPH.

An EXIT statement is not the only statement in a paragraph. The EXIT statement is ignored.

104 SENDING ITEM INVALID OR OMITTED.

A MOVE statement contains an invalid or missing sending operand. Fatal.

105 SENDING ITEM NOT FOLLOWED BY .TO..

A MOVE statement does not have the keyword TO following the sending operand. Fatal.

106 RECEIVING ITEM INVALID OR OMITTED.

A MOVE statement has no valid receiving operand. Fatal.

107 INVALID CLASS FOR DESTINATION FIELD.

The receiving operand of an ADD or SUBTRACT statement is not numeric or numeric edited. Fatal.

110 RELATIVE OR RECORD KEY OR STATUS NAME INVALID.

The name referenced in a RELATIVE KEY, RECORD KEY, ALTERNATE RECORD KEY or FILE STATUS clause is invalid. The clause is ignored.

111 .STOP. SYNTAX ERROR.

The STOP statement is not followed by a literal or the word RUN. Fatal.

112 .SIZE ERROR. STATEMENT INCORRECT.

The word ERROR is not found in the ON SIZE clause. Fatal.

113 .PROCEDURE DIVISION. OMITTED.

The source program does not contain a PROCEDURE DIVISION. Fatal.

114 INTERMEDIATE RESULT TOO LARGE. HIGH ORDER TRUNC.

An arithmetic statement calls for an intermediate result in excess of 18 digits. The intermediate result is truncated on the left to 18 digits, with a possible loss of high-order, non-zero digits at execution time.

115 INTERMEDIATE RESULT TOO LARGE. LOW ORDER TRUNC.

An arithmetic expression calls for an intermediate result in excess of 18 digits. The intermediate result is truncated on the right to 18 digits, with a possible loss of low-order, non-zero digits at execution time.

116 .DIVISION. OMITTED AFTER .PROCEDURE..

The word DIVISION is missing in the PROCEDURE DIVISION header. The error is ignored.

117 TERMINATOR MISSING AFTER DIVISION HEADER.

The period terminator is missing from a division header. The error is ignored.

120 LITERAL INCOMPATIBLE WITH ATTEMPTED USAGE.

Conversion of a literal from one form to another has failed. Fatal.

121 DATANAME MUST FOLLOW .INTO. IN THIS STATEMENT.

A valid data-name is not present following INTO in a STRING or UNSTRING statement. Fatal.

122 NUMERIC SUBJECT OR OBJECT MUST BE INTEGER.

A numeric, non-integer subject or object is invalid in the context of this relation condition. Fatal.

123 OPERANDS CONFLICT IN .SET...TO. STATEMENT.

A SET...TO statement references invalid operands. Fatal.

124 OPERANDS CONFLICT IN .SET ...BY. STATEMENT.

A SET...BY statement references invalid operands. Fatal.

125 ILLEGAL FILENAME LITERAL OR FILENAME DATANAME.

An ASSIGN statement or a VALUE OF ID statement contains an invalid file specification or data-name. The statement is ignored.

126 INVALID SUBJECT OF SIGN CONDITION.

The subject of a sign condition is not a valid arithmetic expression. Fatal.

127 ITEM IN TABLE MAY NOT BE USED AS A SUBSCRIPT.

A data item used as a subscript is itself a table element. Fatal.

130 .POINTER. MUST FOLLOW .WITH. IN THIS STATEMENT.

A STRING or UNSTRING statement has an invalid WITH POINTER phrase. Fatal.

131 RELATIVE KEY INVALID FOR THIS FILE. IGNORED.

A RELATIVE KEY clause has been applied to a file that does not have RELATIVE organization. The RELATIVE KEY clause is ignored.

132 SUBJECT OR OBJECT OMITTED IN RELATION CONDITION.

The subject or object is omitted in a COBOL relation condition. Fatal.

133 UNIDENTIFIABLE WORD FOUND IN SUBSCRIPT.

A subscript list contains a word that is neither a data-name nor a numeric literal. The remainder of the list or sentence is ignored. Fatal.

134 INVALID SUBJECT OR OBJECT IN RELATION CONDITION.

The subject or object of a relation condition is an invalid operand. Fatal.

135 SUBSCRIPTS OMITTED. ASSUME VALUE OF 1.

A reference to a table item contains no subscript list. Literal subscripts of 1 are supplied as defaults.

136 RELATIVE INDEX LITERAL OUT OF RANGE. INDEX USED.

The literal value of a relative index causes an out-of-range reference to the table. The literal value is ignored, and only the index-name is used.

137 SUBSCRIPTS GIVEN WHERE NOT REQUIRED. IGNORED.

A reference is made to a non-table item, and a subscript list follows the reference. The subscript list is ignored.

140 TOO FEW SUBSCRIPTS GIVEN. ASSUME 1 FOR REST.

A reference to a table item contains a subscript list with too few subscripts. Default literal subscripts of 1 are supplied for missing subscripts.

141 TOO MANY SUBSCRIPTS GIVEN. IGNORE EXCESS.

A reference to a table item contains too many subscripts in the subscript list. Extra subscripts are ignored.

142 SUBJECT AND OBJECT USAGE MUST MATCH.

A relation condition between non-numeric operands requires the same usage for both operands. Fatal.

143 ARITHMETIC EXPRESSION REQUIRED IN THIS CONTEXT.

An arithmetic expression is required in the context of the COBOL statement being compiled. The compiler has failed to recognize the arithmetic expression in this context. Fatal.

144 CONDITION EXPRESSION REQUIRED IN THIS CONTEXT.

A condition expression is required in the context of the COBOL statement being compiled. The compiler has failed to recognize the condition expression in this context. Fatal.

145 ILLEGAL OPERAND FOUND IN COBOL EXPRESSION.

An invalid data-name or literal has been found in the COBOL statement being compiled. The class or USAGE of the data item may be invalid here as a reference in an expression. Fatal.

146 OPERATOR IS MISSING IN COBOL EXPRESSION.

An operator is omitted in the specification of this COBOL expression. The compiler cannot recognize this expression as a syntactically valid COBOL expression. Fatal.

147 ABSOLUTE VALUE STORED.

A negative value has been supplied for an unsigned numeric item. The absolute value of the numeric literal is stored in the item.

150 ILLEGAL WORD FOUND AFTER .NOT. IN EXPRESSION.

The compiler has detected an illegal expression operator following a NOT keyword in the COBOL expression being compiled. Fatal.

151 VERB FOUND IN AREA A. ALLOWED.

A statement begins in Area A. The error is ignored.

152 EXPECTED .RELATIVE KEY. DATANAME NOT DEFINED.

The data-name given in a RELATIVE KEY clause has not been defined in the Data Division.

153 .LINAGE. CLAUSE DATAITEM IS TOO LONG.

A data item named in a LINAGE clause is declared in the Data Division with more than four decimal integer positions of precision.

154 PROCEDURE NAME DUPLICATES DATA NAME. ALLOWED.

A procedure name is identical to a data-name. The error is ignored, since there can be no ambiguity in legal references.

155 STATEMENTS FOLLOWING .GO. CAN NEVER BE EXECUTED.

A statement follows an unconditional GO statement. The statements following the GO are compiled, but cannot be executed.

156 NONSEQUENTIAL FILE MAY NOT BE OPTIONAL.

The SELECT statement may specify OPTIONAL only on files with sequential organization. The word OPTIONAL is ignored.

157 FILE HAS IO CONTROL CLAUSE CONFLICTS.

A file is given conflicting clause specifications in the I-O-CONTROL paragraph of the INPUT-OUTPUT SECTION.

160 FILE REQUIRES REL. KEY. TREATED AS SEQ. ACCESS.

A file with relative organization and random or dynamic access has no RELATIVE KEY clause. The access mode is changed to SEQUENTIAL.

161 INVALID INDEX DATAITEM USE IN RELATIONAL.

The compiler detects the invalid use of an index data item reference as the subject or object of a relation condition. Fatal.

162 UNKNOWN WORD. SCAN TO NEXT CLAUSE.

An unknown word is encountered when a clause keyword is expected. All words are ignored up to the next valid clause.

163 CLAUSE DUPLICATED. SECOND OCCURRENCE USED.

A SELECT statement contains two occurrences of the same clause. The second occurrence is used.

164 NO FD FOR THIS SELECT.

The file-name supplied in a SELECT statement is not further described in an FD in the Data Division. The SELECT statement is ignored, causing the file-name to become undefined.

165 DIFFERENT SAME REC. AREAS FOR SAME AREA.

The compiler detects a conflict between the SAME RECORD AREA clause and the SAME AREA clause.

166 .READ. WITHOUT .INVALID KEY. .AT END. OR .USE.

A READ statement contains no conditional clauses, and the file being read has no USE procedure applied to it. Fatal.

167 IO CONTROL CLAUSE HAS FILE WITH NO .SELECT.

An I-O-CONTROL clause references a file-name that was not named in a SELECT statement. The file-name is ignored in the I-O-CONTROL statement.

170 INTEGER OMITTED IN .RESERVE., DEFAULT ASSUMED.

A RESERVE clause fails to specify the number of buffer areas to reserve. The clause is ignored, and the RMS default is used.

171 INVALID SUBJECT OF CLASS CONDITION.

The subject of a class condition is not a data item with an acceptable class. Fatal.

172 VALUE EXCEEDS FIELD CAPACITY. TRUNCATED.

A numeric literal supplied by a VALUE clause exceeds the length of the field. The value is right truncated and stored in the field.

173 NO DATA DIVISION STATEMENTS PROCESSED.

The Data Division contains no valid entries. This is an observation only.

174 INVALID GRP LEV NUM. REST OF RECORD IGNORED.

A level-number is encountered that terminates a previous group item, but does not match any previous group item's level-number. All data entries are skipped until the next 01 level, level indicator or header.

175 INVALID PROCEDURE NAME DEFINITION IN AREA A.

The compiler detects source text in Area A of the Procedure Division that does not conform to the rules for the definition of a legitimate paragraph or section name. Source text found in Area A of the Procedure Division is interpreted by the compiler as a user attempt to define a new paragraph or section name. The compiler supplies a system-defined procedure name and proceeds with the processing of the source line text containing the invalid Area A text. The system-defined procedure name is transparent and, thus, inaccessible to the user.

176 MISSING QUOTE ON CONTINUE LINE. QUOTE ASSUMED.

A non-numeric literal is continued, but the first non-space character is not a quote. The error is ignored by assuming a quote in front of the first non-space character.

177 COMPARISON OF LITERALS IS NOT PERMITTED.

A relation condition has a literal as both subject and object. Fatal.

200 COPY IGNORED WITHIN LIBRARY TEXT.

A COPY statement is encountered within library text. The COPY statement is ignored.

201 INVALID FILENAME ON COPY. COPY IGNORED.

A COPY statement supplies a file specification that is invalid. The COPY statement is ignored.

202 COPY FILENAME NOT FOUND.

A COPY statement supplies a valid file specification, but the file cannot be found on the specified device. The COPY statement is ignored.

203 PERIOD OMITTED AFTER .DECLARATIVES..

The word DECLARATIVES is not followed by a period. The error is ignored.

204 .DECLARATIVES. OMITTED FROM .END. STATEMENT.

The word END is not followed by DECLARATIVES. END DECLARATIVES is assumed.

205 PERIOD OMITTED AFTER .END DECLARATIVES..

The words END DECLARATIVES are not followed by a period. The error is ignored.

206 SOURCE PROGRAM ENDS IN DECLARATIVES.

The end of the source program occurs in the Declaratives area. Fatal.

207 DATANAME MUST FOLLOW .WITH POINTER. PHRASE.

A STRING or UNSTRING statement contains an invalid WITH POINTER phrase. Fatal.

210 .OVERFLOW. MUST FOLLOW .ON. IN THIS STATEMENT.

A STRING or UNSTRING statement contains an invalid ON OVERFLOW phrase. Fatal.

211 ILLEGAL SENDING FIELD DATANAME IN .UNSTRING.

The sending field of an UNSTRING statement has an invalid class. Fatal.

212 ILLEGAL SYNTAX IN .UNSTRING. STATEMENT.

An UNSTRING statement has invalid syntax. Fatal.

213 MULTIPLE SIGN CLAUSES ON THIS ITEM.

More than one SIGN clause appears in a data description. (SEPARATE must follow LEADING or TRAILING.) The second clause is used.

214 ILLEGAL SYNTAX IN COBOL EXPRESSION.

The compiler detects a syntax error of a general nature in the COBOL expression being compiled. Fatal.

215 SIGN CLAUSE ON NONNUMERIC ITEM.

A SIGN clause appears in a non-numeric data description. The SIGN clause is ignored.

216 SIGN CLAUSE APPLIED TO NONDISPLAY ITEM.

A SIGN clause appears in a numeric data description with usage other than DISPLAY. The SIGN clause is ignored.

217 SIGN CLAUSE APPLIED TO UNSIGNED DATAITEM.

A SIGN clause appears in a numeric data description that has no "S" in its PICTURE string. The SIGN clause is ignored.

220 ILLEGAL DELIMITING DATA ITEM IN .UNSTRING.

An UNSTRING statement references an invalid delimiter. Fatal.

221 .ALL. FIGURATIVE CONSTANT ILLEGAL IN .UNSTRING.

An UNSTRING statement contains an ALL literal reference. Fatal.

222 ILLEGAL RECEIVING DATANAME IN .UNSTRING.

An UNSTRING statement references a receiving data item that is invalid. Fatal.

223 .DELIMITED. CLAUSE REQUIRED IN THIS .UNSTRING.

An UNSTRING statement contains no DELIMITED BY clause. Fatal.

224 DATANAME MUST FOLLOW .DELIMITER IN. PHRASE.

An UNSTRING statement contains a DELIMITER IN phrase with an illegal reference. Fatal.

225 ILLEGAL DATANAME FOLLOWS .DELIMITER IN. PHRASE.

An UNSTRING statement contains a DELIMITER IN phrase referencing a data item that is invalid. Fatal.

226 DATANAME MUST FOLLOW .COUNT IN. PHRASE.

An UNSTRING statement contains a COUNT IN phrase with an illegal reference. Fatal.

227 ILLEGAL DATANAME FOLLOWS .COUNT IN. PHRASE.

An UNSTRING statement contains a COUNT IN phrase that references an invalid data item. Fatal.

230 DATANAME MUST FOLLOW .TALLYING IN. PHRASE.

An UNSTRING statement contains a TALLYING phrase with an illegal reference. Fatal.

231 ILLEGAL DATANAME FOLLOWS .TALLYING IN. PHRASE.

An UNSTRING statement contains a TALLYING phrase referencing a data item that is invalid. Fatal.

232 DATANAME MUST FOLLOW .INSPECT. VERB.

A valid data-name reference does not follow the INSPECT keyword. Fatal.

233 ILLEGAL DATANAME FOLLOWS .INSPECT. VERB.

An INSPECT statement references a data item that is invalid. Fatal.

234 ILLEGAL DATANAME PRECEDES .FOR. IN .INSPECT.

An INSPECT...TALLYING statement references a tally data item that is invalid. Fatal.

235 .FOR. OMITTED IN .INSPECT. STATEMENT.

An INSPECT...TALLYING statement has invalid syntax. Fatal.

236 DATANAME MUST FOLLOW .TALLYING. PHRASE.

An INSPECT...TALLYING statement does not reference a tally data-name. Fatal.

237 ILLEGAL WORD FOLLOWS .FOR. IN .INSPECT.

An INSPECT...TALLYING statement does not state a valid search condition. Fatal.

240 DATAITEM OMITTED AFTER .ALL. .LEADING. OR .FIRST.

An INSPECT statement does not reference a valid search argument. Fatal.

241 .ALL. FIGURATIVE CONSTANT ILLEGAL IN .INSPECT.

An ALL literal appears in an INSPECT statement. Fatal.

242 ILLEGAL DATANAME FOLLOWS .ALL. OR .LEADING.

An INSPECT statement does not reference a valid search argument. Fatal.

243 ILLEGAL DATANAME FOLLOWS .BEFORE. OR .AFTER.

An INSPECT statement does not reference a valid delimiter in the BEFORE/AFTER phrase. Fatal.

244 ILLEGAL DATANAME FOLLOWS .BY.

An INSPECT statement does not reference a valid replacement argument. Fatal.

245 ILLEGAL DATANAME PRECEDES .BY.

An INSPECT statement does not reference a legal data-name or literal preceding the BY phrase. Fatal.

246 DATAITEM OMITTED IN .BEFORE. OR .AFTER. PHRASE.

An INSPECT statement does not reference a legal data-name or literal after the BEFORE or AFTER phrase. Fatal.

247 ILLEGAL SYNTAX IN .INSPECT. STATEMENT.

Both the TALLYING and REPLACING keywords are missing in the INSPECT statement. Fatal.

250 .BY. MUST FOLLOW .CHARACTERS. IN REPLACING LIST.

The INSPECT...REPLACING statement must have CHARACTERS BY phrase completely specified. Fatal.

251 DATAITEM OMITTED AFTER .BY. IN .INSPECT.

The INSPECT...REPLACING statement does not reference a legal data-name or literal after BY. Fatal.

252 DATAITEM FOLLOWING .BY. EXCEEDS 1 CHARACTER.

In an INSPECT...REPLACING statement, when: 1) the CHARACTERS BY phrase is specified, or 2) a figurative constant preceding the BY keyword of the ALL, LEADING, or FIRST phrase is specified, the data-name or literal after the BY keyword must be defined as one character in length. Fatal.

253 DATAITEMS BEFORE AND AFTER .BY. UNEQUAL IN SIZE.

In an INSPECT...REPLACING statement, the data items before and after the BY keyword of the ALL, LEADING, or FIRST phrase must be equal in length. Fatal.

254 .BEFORE. OR .AFTER. OPERAND EXCEEDS 1 CHARACTER.

In an INSPECT...REPLACING CHARACTERS BY statement, the data-name or literal following the BEFORE or AFTER keyword must be one character in length. Fatal.

255 ILLEGAL WORD FOLLOWS .REPLACING. IN .INSPECT.

A legal keyword was not recognized following REPLACING in the INSPECT statement. Fatal.

256 .BY. OMITTED AFTER REPLACING COMPARISON OPERAND.

The keyword BY is omitted in the ALL, LEADING, or FIRST phrase where it separates operands to be compared. Fatal.

257 TOO MANY RIGHT PARENTHESES IN COBOL EXPRESSION.

The compiler detects an excess of right parentheses in the COBOL expression being compiled. Parentheses must be specified in balanced pairs; that is, a left parenthesis must exist for each right parenthesis specified. Fatal.

260 TOO MANY LEFT PARENTHESES IN COBOL EXPRESSION.

The compiler detects an excess of left parentheses in the COBOL expression being compiled. Parentheses must be specified in balanced pairs; that is, a right parenthesis must exist for each left parenthesis specified. Fatal.

261 MISSING OPERAND IN ARITHMETIC EXPRESSION.

An operand is omitted in a COBOL arithmetic expression. Fatal.

262 ILLEGAL OPERAND IN ARITHMETIC EXPRESSION.

The compiler detects an illegal operand in a COBOL arithmetic expression. The class or usage of the operand may be invalid in the context as a reference in an arithmetic expression. Fatal.

263 NONINTEGER EXPONENT FOUND IN COBOL EXPRESSION.

The compiler detects a non-integer, numeric exponent in a COBOL arithmetic expression. The arithmetic expression is considered invalid. Fatal.

264 SUBJECT OMITTED IN CLASS CONDITION.

The compiler detects the omission of the subject in a NUMERIC or ALPHABETIC class condition. Fatal.

265 SUBJECT OMITTED IN SIGN CONDITION.

The compiler detects the omission of the subject in a sign condition. Fatal.

266 OPERAND MISSING IN COMPLEX CONDITION.

The compiler detects the omission of an operand in an AND or OR complex condition. Fatal.

267 INVALID OPERAND IN COMPLEX EXPRESSION.

The compiler detects a complex condition operand that is not a simple condition. combined condition, or complex condition. Fatal.

270 ILLEGAL SYNTAX IN NEGATED SIMPLE CONDITION.

The compiler detects illegal syntax in a COBOL negated simple condition. Fatal.

271 INVALID NEGATED SIMPLE CONDITION.

The compiler detects the application of the NOT keyword to an invalid simple condition. Fatal.

272 ILLEGAL SYNTAX IN .COMPUTE. STATEMENT.

The compiler detects illegal syntax in a COMPUTE statement. The left side of the assignment symbol or the assignment symbol itself may have been omitted. Fatal.

273 .AT END. ILLEGAL FOR RANDOM .READ.

The file is specified with either ACCESS RANDOM or ACCESS DYNAMIC without the word NEXT being included in the READ statement. The AT END clause is treated as an INVALID KEY clause.

274 INVALID KEY ILLEGAL FOR SEQUENTIAL .READ.

Either the file has ACCESS SEQUENTIAL or the READ statement contains the word NEXT. In either case, the INVALID KEY clause is illegal. It is treated as an AT END clause.

275 INDEX DATA ITEM ILLEGAL AS INDEX ON TABLE.

An index data item is used as an index for a table. The index data item reference is ignored. A literal subscript of 1 replaces the index data item reference.

276 INDEX NAME NOT DEFINED FOR THIS TABLE.

An index-name used in a subscript list either is not defined for this table or appears in the wrong logical position of the subscript list for this table. The index-name is ignored and a default value of 1 is assumed as the subscript.

277 RELATIVE INDEX IS INVALID.

The literal component of a relative index is zero or less in value, or is an invalid word. Relative indexing is ignored and only the index-name is used.

300 PROGRAM NAME OMITTED AFTER .CALL. VERB.

The program-name is omitted after the key word CALL. Fatal.

301 LINAGE 0 OR LESS THAN FOOTING.

The LINAGE clause must specify a page body of at least one line, and the page body size must be equal to or greater than the footing size specified in the FOOTING phrase.

302 FILE CLOSED BUT NOT OPENED.

A CLOSE statement was encountered for a file that is not opened in this program. Fatal.

303 PRINT CONTROL ON NON SEQUENTIAL FILE. IGNORED.

An APPLY PRINT-CONTROL clause references a file that does not have SEQUENTIAL organization. The file-name is ignored in the APPLY clause.

304 DATANAME OMITTED IN .KEY IS. PHRASE.

The KEY IS phrase of the START statement is not followed by a data-name. The prime RECORD KEY data-name is assumed present.

305 SECTION OR PARAGRAPH NAME MISSING.

The Procedure Division does not start with a section or paragraph name, or a section header is not followed by a paragraph name. Fatal.

306 .PROCEDURE. MISSING IN .USE. STATEMENT. ASSUMED.

The keyword PROCEDURE is missing in the USE statement. It is assumed and processing is continued.

307 .START. WITHOUT .INVALID KEY. OR .USE.

The INVALID KEY option is missing from the START statement, or no USE procedure is declared for the referenced file. Fatal.

310 .WRITE. WITHOUT .INVALID KEY. OR .USE.

The INVALID KEY option is missing from the WRITE statement, or no USE procedure is declared for the referenced file. Fatal.

311 DATA DIVISION MUCH TOO LARGE.

Too much buffer space is being used for the files in this program. Too many files are declared to be OPEN simultaneously. Fatal.

312 .REDEFINES. SPECIFIES INVALID REDEFINITION.

The compiler detects the invalid application of REDEFINES to a data description entry that contributes new character positions between the data description entry containing the REDEFINES clause and the item being redefined. Also, the source of error may be the definition of another data description entry with a lower level number appearing between the data description entry containing the REDEFINES clause and the item being redefined. The compiler ignores the REDEFINES clause and continues processing the data description entry.

313 ILLEGAL TO REDEFINE ANOTHER REDEFINITION.

The REDEFINES clause specifies the redefinition of a data item whose data description entry contains a REDEFINES clause itself. The compiler ignores the REDEFINES clause and continues processing the data description entry.

314 ILLEGAL TO REDEFINE A COBOL TABLE.

The REDEFINES clause specifies the redefinition of a data item whose data description entry contains an OCCURS clause. The compiler ignores the REDEFINES clause and continues processing the data description entry.

315 .REDEFINES. APPLIED TO VARIABLE LENGTH DATAITEM.

The compiler detects an application of the REDEFINES clause to a data item whose length is variable at run time because it has a subordinate data item whose data description entry contains an OCCURS DEPENDING ON clause. The application of the REDEFINES clause to such a data item is syntactically invalid. The compiler ignores the REDEFINES clause and continues processing the data description entry.

316 .OCCURS DEPENDING ON. ILLEGAL IN REDEFINITION.

The compiler detects a redefinition that contains a data description entry declared with an OCCURS DEPENDING ON clause. The OCCURS DEPENDING ON clause causes the redefinition to contain a data item whose length is variable at run time. The DEPENDING ON phrase is ignored and processing continues.

317 PICTURE EXCEEDS 30 CHARACTERS. PIC X ASSUMED.

The unexpanded PICTURE string exceeds 30 characters in length. The compiler ignores the user-supplied PICTURE and treats the data item as alphanumeric with a "PICTURE X" declaration.

320 FILENAME MUST FOLLOW .CLOSE VERB.

The data item following the CLOSE verb was not a file-name. Fatal.

321 .NO. MUST FOLLOW .WITH. IT IS ASSUMED.

The keyword NO is missing in the WITH NO REWIND phrase of the CLOSE statement. NO is assumed present.

322 .REWIND. MUST FOLLOW .NO. IT IS ASSUMED.

The WITH NO REWIND phrase of the CLOSE statement must be completely specified. It is assumed present.

323 .REMOVAL. MUST FOLLOW .FOR. IT IS ASSUMED.

The FOR REMOVAL phrase of the CLOSE statement must be completely specified. It is assumed present.

324 .LOCK. OMITTED AFTER .WITH. IT IS ASSUMED.

The keyword WITH in a CLOSE statement is recognized but is not followed by one of the keywords NO or LOCK. The WITH LOCK phrase is assumed present.

325 DATANAME SPECIFIED WHERE FILENAME EXPECTED.

The name used in an I/O verb to reference a file was not a file name but was some other data-name. Fatal.

326 FILENAME MUST FOLLOW MODE SPEC. IN .OPEN.

The OPEN statement does not reference a valid file name where a file-name reference is expected. Fatal.

327 ILLEGAL MODE SPECIFIED AFTER .OPEN. VERB.

One of the OPEN mode keywords INPUT, OUTPUT, I-O, or EXTEND is required immediately after the OPEN verb. Fatal.

330 .END. MUST FOLLOW .AT.. IT IS ASSUMED.

The keyword END was omitted in the AT END phrase of the READ statement. The AT END phrase is assumed present.

331 FILENAME MUST FOLLOW .READ. VERB.

Either the file-name was omitted following the READ verb or the data item following the READ verb is not a valid file-name reference. Fatal.

332 DATANAME OMITTED AFTER .INTO. IN .READ.

The data-name reference following the INTO keyword of the READ statement was omitted. Fatal.

333 RECORDNAME MUST FOLLOW .WRITE. OR .REWRITE.

The 01 record-name reference immediately following the WRITE or REWRITE verb was omitted. Fatal.

334 STATEMENT IGNORED DUE TO ILLEGAL RECORDNAME.

The data-name immediately following the WRITE or REWRITE verb is not a valid 01 record-name reference. Fatal.

335 .ADVANCING. OPTION OMITTED IN .WRITE. 1 ASSUMED.

A data-name reference, numeric integer literal reference, or the keyword PAGE was not recognized in the BEFORE/AFTER ADVANCING phrase of the WRITE statement. A numeric integer literal value of 1 is assumed.

336 .EOP. MUST FOLLOW .AT.. IT IS ASSUMED.

The keyword EOP was omitted in the AT EOP phrase of the WRITE statement. The AT EOP phrase is assumed present.

337 DATANAME OMITTED AFTER .FROM.

The data-name reference following the FROM keyword of the WRITE or REWRITE statement was omitted. Fatal.

340 .ADVANCING. INTEGER TOO BIG. TRUNCATED TO 63.

The numeric integer in the BEFORE/AFTER ADVANCING phrase of the WRITE statement is greater than 63. 63 is assumed.

341 .NO REWIND. ILLEGAL WITH .IO. OR .EXTEND. MODE.

An OPEN statement with the I-O or EXTEND mode specified cannot have the NO REWIND phrase also specified. Fatal.

342 ILLEGAL .ADVANCING. DATANAME. 1 IS ASSUMED

The data-name in the BEFORE/AFTER ADVANCING phrase of the WRITE statement is not an elementary numeric integer data-name reference. A numeric integer literal value of 1 is assumed.

343 FILENAME MUST FOLLOW .DELETE. VERB.

Either the file-name was omitted following the DELETE verb or the data item following the DELETE verb is not a valid file-name reference. Fatal.

344 FILENAME MUST FOLLOW .START. VERB.

Either the file name was omitted following the START verb or the data item following the START verb is not a valid file name reference. Fatal.

345 .LESS. OMITTED AFTER .NOT. IN .START. ASSUMED.

The keyword LESS is omitted after NOT in the relational condition of the START statement. LESS is assumed present.

346 DATANAME OMITTED IN .KEY IS. PHRASE. ASSUMED.

The RELATIVE KEY data-name for the referenced file was omitted in the KEY IS phrase of the START statement. The RELATIVE KEY data-name is assumed present.

347 RELATIONAL WORD OMITTED AFTER .KEY IS. PHRASE.

None of the relational keywords EQUAL, GREATER, or NOT was recognized following the KEY IS phrase of the START statement. Fatal.

350 TERMINATOR IGNORED IN .IO CONTROL. PARAGRAPH.

A clause is terminated by a period, but a header does not follow in Area A. The period is ignored. The compiler assumes it is still in the I-O-CONTROL paragraph.

351 TERMINATOR IGNORED IN .SPECIAL NAMES. PARAGRAPH

A clause is terminated by a period, but is not followed by a header in Area A. The period is ignored, and the compiler continues processing the SPECIAL-NAMES paragraph.

352 .NATIVE. MISSING IN SPECIAL NAMES CLAUSE.

The alphabet-name clause does not contain NATIVE or STANDARD-1. The alphabet-name clause is ignored.

353 SYNTAX ERROR IN .OBJECT COMPUTER. PARAGRAPH.

The OBJECT-COMPUTER paragraph contains an unrecognizable word. The compiler scans over all words until a word is found in Area A.

354 TERMINATOR OMITTED IN .OBJECT COMPUTER. PARA.

The OBJECT-COMPUTER paragraph is not terminated by a period. The compiler scans over all words until a word is found in Area A.

355 DATANAME FOLLOWING .KEY IS. PHRASE IS ILLEGAL.

The data-name following the KEY IS phrase of the START statement is not a RECORD KEY associated with the referenced indexed file, nor is it subordinate to a RECORD KEY whose leftmost character position corresponds to its own leftmost character position. Fatal.

356 INVALID USAGE ON CONDITIONAL VARIABLE.

The level 88 condition variable cannot be defined as USAGE INDEX.

357 ILLEGAL SEPARATOR IN COBOL STATEMENT. IGNORED.

An illegal character was detected between two consecutive words of a COBOL statement. The illegal character is ignored.

360 ILLEGAL CHARACTER FOUND WITHIN A COBOL WORD.

Illegal characters were found in an alphanumeric COBOL word, but not in an alphanumeric literal. The illegal characters are replaced by dollar signs in the internal representation of the COBOL word.

361 UNRECOGNIZABLE TEXT FOUND IN COBOL STATEMENT.

In scanning the source text, the compiler was unable to recognize an alphanumeric COBOL word (a keyword or user-defined word), an alphanumeric literal, or a numeric literal. The error is not internally corrected and usually will cause further error messages.

362 COBOL WORD BEGINS WITH OR ENDS IN HYPHEN.

In attempting to recognize a keyword or user-defined word, the compiler has detected that the COBOL word begins or ends with a hyphen.

363 NONNUMERIC LITERAL TOO LONG. TRUNCATED TO MAX.

An alphanumeric literal greater than 132 characters in length is detected. The literal is truncated on the right, retaining the first 132 characters as the literal.

364 COBOL SOURCE LINE TOO LONG. TRUNCATED TO MAX.

The indicated COBOL source line contains more than 65 characters in terminal format. The excess characters are ignored, and only those characters in the printed COBOL source line are retained.

365 .BY. OMITTED IN REPLACING OPTION. COPY IGNORED.

The keyword BY was not found in a COPY...REPLACING statement. The statement is ignored.

366 TERMINATOR OMITTED IN .COPY. IT IS ASSUMED.

The required period terminating the COPY statement is omitted. It is assumed present.

367 .LINAGE. CLAUSE DATANAME MUST BE AN INTEGER.

A data-name referenced in the LINAGE clause of the FILE SECTION is defined with decimal places in the WORKING-STORAGE SECTION.

370 .LINAGE.CLAUSE DATANAME MUST BE UNSIGNED.

A numeric data-name referenced in the LINAGE clause of the FILE SECTION is defined as a signed data item in the WORKING-STORAGE SECTION.

371 POSSIBLE HIGH ORDER RECEIVING FIELD TRUNCATION.

Truncation of high-order information during a MOVE or an arithmetic operation upon a receiving field is possible. This truncation could cause unexpected results, and the message should not be ignored.

372 POSSIBLE LOW ORDER RECEIVING FIELD TRUNCATION.

Truncation of low-order information during a MOVE or an arithmetic operation upon a receiving field is possible. This truncation could cause unexpected results, and the message should not be ignored.

373 FD HEADER NOT FOLLOWED BY AN AREA A WORD.

The word following the PROCEDURE DIVISION header does not begin in Area A. The compiler scans over all words until a word is found in Area A.

374 OPEN OPTIONAL FILES ONLY IN .INPUT. MODE.

An OPTIONAL file can be OPENed in INPUT mode only. The compiler assumes that the OPTIONAL file is OPENed in INPUT mode.

375 EXPECTED .FILE STATUS. DATANAME NOT DEFINED.

A data-name referenced in a FILE STATUS phrase of a SELECT clause in the FILE-CONTROL paragraph is not defined in the WORKING-STORAGE SECTION of the DATA DIVISION.

376 EXPECTED .VALUE OF ID. DATANAME NOT DEFINED.

The data-name referenced in a VALUE OF ID clause of an FD is not defined in the WORKING-STORAGE SECTION of the DATA DIVISION. Fatal.

377 EXPECTED .LINAGE. CLAUSE DATANAME NOT DEFINED.

A data-name referenced in the LINAGE clause of the FILE SECTION is not defined in the WORKING-STORAGE SECTION of the DATA DIVISION.

400 .RELATIVE KEY. DATANAME HAS INVALID CLASS.

A data-name referenced in a RELATIVE KEY phrase of a SELECT clause in the FILE-CONTROL paragraph is defined with non-numeric class in the WORKING-STORAGE SECTION.

401 .RELATIVE KEY. DATANAME HAS INVALID CLASS.

A data-name referenced in a RELATIVE KEY phrase of a SELECT clause must not be defined with INDEX usage in the WORKING-STORAGE SECTION.

402 .RELATIVE KEY. DATAITEM IS TOO LONG.

A numeric integer data-name referenced in a RELATIVE KEY phrase is defined with more than eight digits of precision in the WORKING-STORAGE SECTION.

403 .RELATIVE KEY. DATANAME MUST BE AN INTEGER.

A numeric data-name referenced in a RELATIVE KEY phrase is defined with decimal places in the WORKING-STORAGE SECTION.

404 .FILE STATUS. DATANAME HAS INVALID CLASS.

A data-name referenced in a the FILE STATUS phrase of a SELECT clause must be defined in with DISPLAY usage in the WORKING-STORAGE SECTION.

405 .FILE STATUS. DATA NAME HAS INVALID USAGE.

A data-name referenced in a FILE STATUS phrase of a SELECT clause is defined with DISPLAY USAGE in the WORKING-STORAGE SECTION.

406 LENGTH OF .FILE STATUS. DATAITEM IS ILLEGAL.

An alphanumeric data-name referenced in a FILE STATUS phrase of a SELECT clause must be defined in the WORKING-STORAGE SECTION as an alphanumeric variable consisting of two characters.

407 .VALUE OF ID. DATANAME HAS INVALID CLASS.

A data-name referenced in a VALUE OF ID clause of an FD is defined with non-alphanumeric class in the WORKING-STORAGE SECTION.

410 .VALUE OF ID. DATANAME HAS INVALID USAGE.

A data-name referenced in a VALUE OF ID clause of an FD must be defined with DISPLAY usage in the WORKING-STORAGE SECTION.

411 LENGTH OF .VALUE OF ID. DATAITEM IS ILLEGAL.

An alphanumeric data-name referenced in a VALUE OF ID clause of an FD must be defined in the WORKING-STORAGE SECTION as an alphanumeric variable whose length, L, falls in the range 9<=L<=150 characters. Fatal.

412 .LINAGE. CLAUSE DATANAME HAS INVALID CLASS.

A data-name referenced in the LINAGE clause of the FILE SECTION is defined with non-numeric class in the WORKING-STORAGE SECTION.

413 .LINAGE. CLAUSE DATANAME HAS INVALID USAGE.

A data-name referenced in the LINAGE clause of the FILE SECTION must be defined with COMPUTATIONAL USAGE in the WORKING-STORAGE SECTION.

414 INVALID RECEIVING OPERAND IN .SET.. IGNORED.

A receiving operand of a SET statement is invalid. Fatal.

415 NO RECEIVING OPERAND SPECIFIED IN .SET..

No receiving operands are specified in a SET statement. Fatal.

416 OMITTED OR ILLEGAL OPERAND AFTER .TO. IN .SET..

A SET statement has no valid sending operand. Fatal.

417 ILLEGAL SYNTAX IN .SET. STATEMENT.

The words TO, UP or DOWN do not follow the receiving operands of a SET statement. Fatal.

420 .BY. MUST FOLLOW .UP. OR .DOWN.. ASSUMED.

The keyword BY does not follow the word UP or DOWN in a SET statement. BY is assumed present.

421 OMITTED OR ILLEGAL OPERAND AFTER .BY. IN .SET..

The operand following the UP BY or DOWN BY phrase in a SET statement is invalid or omitted. Fatal.

422 NO OPERANDS SPECIFIED

No operands were recognized following the keyword DISPLAY. Fatal.

423 SETTING INDEX NAME OUT OF RANGE. .SET. IGNORED.

A SET statement is attempting to set an index name using a literal that is too large. Fatal.

424 .IF. TRUE PATH OMITTED. ASSUME .NEXT SENTENCE..

The true path code is omitted from the IF statement. NEXT SENTENCE is assumed as the true path of the IF statement.

425 CONFLICTING SIGN SYMBOLS IN PICTURE STRING.

The compiler recognizes both the + and - sign symbols in this PICTURE string. The compiler ignores the user-supplied PICTURE and treats the data item as alphanumeric with a "PICTURE X" declaration.

426 ZERO SUPPRESSION CONFLICTS IN PICTURE STRING.

The compiler recognizes both the Z and * zero suppression symbols in this PICTURE string. The compiler ignores the user-supplied PICTURE and treats the data item as alphanumeric with a "PICTURE X" declaration.

427 ILLEGAL CHARACTER IN THE PICTURE STRING.

A character that is not in the PICTURE string character set is recognized in this PICTURE by the compiler. The compiler ignores the user-supplied PICTURE and treats the data item as alphanumeric with a "PICTURE X" declaration.

430 .BLANK WHEN ZERO. CONFLICTS WITH ZERO SUPPRESS.

A BLANK WHEN ZERO clause is recognized with a zero suppression field specified in the PICTURE string. The compiler ignores the BLANK WHEN ZERO clause and continues with its processing.

431 PARENTHESIZED SPECIFIER EXCEEDS 18 DIGITS.

The specification contained inside the parentheses of a PICTURE string exceeds 18 digits in length. The compiler ignores the user-supplied PICTURE and treats the data item as alphanumeric with a "PICTURE X" declaration.

432 SPECIFIER MISSING INSIDE PARENTHESES.

The specification contained inside parentheses of a PICTURE string is missing. The compiler ignores the user-supplied PICTURE and treats the data item as alphanumeric with a "PICTURE X" declaration.

433 ILLEGAL SYMBOL PRECEDES LEFT PAREN. IN PICTURE.

The compiler recognizes an S, V, CR, DB, or "." character preceding a left parenthesis in a PICTURE string. The error is ignored and processing continues.

434 TERMINATOR OMITTED IN .NOTE. PARAGRAPH.

The compiler detected a NOTE paragraph that does not end with a period.

435 INVALID OPERAND IN .VARYING. OR .AFTER. PHRASE.

The expected operand is not a valid name reference in the VARYING or AFTER phrase of this PERFORM VARYING statement. Fatal.

436 INVALID OPERAND IN .FROM. OR .BY. PHRASE.

The FROM or BY phrase of a PERFORM VARYING statement does not contain a valid operand reference. Fatal.

437 TOO MANY .AFTER. PHRASES IN .PERFORM. STATEMENT.

The compiler detects more than two AFTER phrases in the PERFORM VARYING statement being compiled. Fatal.

440 .FROM. OR .BY. OR .UNTIL. MISSING IN PERFORM.

The compiler detects the omission of the keywords FROM, BY, or UNTIL in the PERFORM VARYING statement. Fatal.

441 ILLEGAL CONDITION EXPRESSION IN THE PERFORM.

The compiler detects an invalid condition expression in the PERFORM statement. Fatal.

442 NONPOSITIVE LITERAL IN .FROM. OR .BY. PHRASE.

The compiler detects a non-positive, numeric integer literal in this PERFORM statement. Fatal.

443 INVALID RELATION CONDITION IN .SEARCH ALL.

The compiler detects either a syntax error or an invalid operand in the restricted form of a relation condition in the SEARCH ALL statement. Fatal.

444 NONINTEGER DATA CONFLICTS WITH INDEXNAME USAGE.

The compiler detects a non-integer data item reference in a PERFORM VARYING statement in which the VARYING, AFTER, and/or FROM phrase contains an index-name reference. Fatal.

445 IMPLICIT REFERENCE TO BAD CONDITION VALUES.

Through a reference to a condition-name, the compiler detects a reference to an associated condition-value that is improperly declared in the Data Division. Fatal.

446 IMPLICIT REFERENCE TO BAD CONDITION VARIABLE.

Through a reference to a condition-name, the compiler detects that the associated condition-variable is improperly declared in the Data Division. Fatal.

447 TOO MANY NAMES IN COBOL PROGRAM. RECOMPILE.

The COBOL program being compiled has too many data-names or procedure-names. This condition has caused a compiler table to overflow, aborting the compilation.

450 REFERENCE TO UNDEFINED DATANAME OR ILLEGAL SYNTAX

The COBOL statement being compiled contains a reference to an undefined data-name. The compiler ignores the reference. This diagnostic may be issued in conjunction with other diagnostics for the erroneous statement. Fatal.

451 QUALIFIED REFERENCE ILLEGAL IN THIS CONTEXT.

The compiler detects a qualified reference in a context in which an unqualified reference is required. The compiler permits the qualified reference in this context and continues with the compilation of the statement containing the reference.

452 QUALIFIER OMITTED IN QUALIFIED REFERENCE.

A data-name is omitted after the keyword OF or IN in a qualified reference in the COBOL statement being compiled. The reference is ignored. This diagnostic may be issued in conjunction with other diagnostics for the statement in error.

453 TOO MANY QUALIFIERS IN QUALIFIED REFERENCE.

The compiler detects more than 48 qualifiers in a qualified reference. The excess qualifiers are ignored in the reference.

454 UNDEFINED QUALIFIER IN QUALIFIED REFERENCE.

The compiler detects a qualified reference in which a qualifier is a reference to an undefined data-name. The compiler ignores the entire qualified reference. This diagnostic may be issued in conjunction with other diagnostics for the erroneous statement containing the reference.

455 COBOL STATEMENT CONTAINS AMBIGUOUS REFERENCE.

The compiler detects a reference to COBOL data that is not uniquely referenceable through qualification. The compiler uses a reference that satisfies the reference in the text of the COBOL program. This diagnostic may be issued in conjunction with other diagnostics for the statement in error.

456 DATANAME REFERENCE EXPECTED IN THIS CONTEXT.

The compiler detects a reference to a data item that is not alphabetic, numeric, alphanumeric-edited, alphanumeric, or numeric-edited. The context of this reference requires that the reference be to one of these classes of data items. This diagnostic may be issued in conjunction with other diagnostics for the statement in error.

457 ILLEGAL REFERENCE DETECTED IN THIS CONTEXT.

The compiler detects a reference to an item that is invalid in the context of its usage. This diagnostic may be issued in conjunction with other diagnostics for the statement in error. Fatal.

460 PARENTHESIZED SPECIFIER LARGER THAN 65387

The specification contained in parentheses in a PICTURE string is greater than 65387. The compiler assumes 65387 and continues processing.

461 EXTRA OPENING QUOTE ON LITERAL IS IGNORED.

The compiler detects a superfluous quote at the beginning of a non-numeric literal specification. The compiler ignores the extra quote and continues processing the non-numeric literal.

462 PROGRAM NAME MUST BE A NONNUMERIC LITERAL.

The program-name literal following the key word CALL is not a nonnumeric literal. Fatal.

464 LITERALS ARE ILLEGAL IN ARGUMENT LIST OF .CALL..

Literals are not allowed in the argument list of a CALL statement. Fatal.

465 ARGUMENT LIST OMITTED AFTER .USING. IN .CALL..

The required argument list is missing after the key word USING in the CALL statement. Fatal.

470 ILLEGAL SYNTAX IN .CODE SET. CLAUSE. IGNORED.

A valid alphabet-name reference is omitted in the CODE-SET clause. The compiler ignores the CODE-SET clause and continues to process the remainder of the FD.

471 DATANAME IN .KEY IS. PHRASE NOT ALPHANUMERIC.

The data-name following the KEY IS phrase in a START statement referencing an indexed file must be alphanumeric. Fatal.

472 .RECORD KEY. DATAITEM LENGTH GREATER THAN 255.

A data-name referenced in a RECORD KEY or ALTERNATE RECORD KEY phrase of a SELECT clause in the FILE-CONTROL paragraph must be defined in the FILE SECTION as an item whose length is less than or equal to 255.

473 DATANAME IN .KEY IS PHRASE IS SUBSCRIPTED OR INDEX.

The data-name following the KEY IS phrase in a READ or START statement referencing an indexed file must not be subscripted or indexed. Fatal.

474 .RECORD KEY. DATAITEM MUST NOT BE A COBOL TABLE.

A data-name referenced in a RECORD KEY or ALTERNATE RECORD KEY phrase of a SELECT clause in the FILE-CONTROL paragraph must not be defined in the FILE SECTION with an OCCURS clause or be subordinate to an item with an OCCURS clause.

475 .RECORD. OMITTED FROM .ALTERNATE RECORD. ASSUMED.

The reserved word RECORD is missing from the ALTERNATE RECORD KEY clause. The error is ignored.

476 UNDEFINED .ALTERNATE RECORD KEY. DATANAME.

The data-name given in an ALTERNATE RECORD KEY clause has not been defined in the Data Division.

477 .ALTERNATE RECORD KEY. CLAUSES ARE SEPARATED.

In the SELECT statement the ALTERNATE RECORD KEY clauses are interleaved among the other clauses. The ALTERNATE RECORD KEY clauses should follow one another with no intervening clauses. This error is ignored.

500 LINKAGE SECTION ITEM APPEARS TWICE IN .USING..

A LINKAGE SECTION data item must not appear more than once in the USING phrase of a PROCEDURE DIVISION USING header. Fatal.

501 ILLEGAL .SEGMENT-LIMIT. VALUE IGNORED.

The segment-limit is not a numeric literal or is a numeric literal whose value is outside of allowed segment-limit range.

502 INTEGER 1 BEYOND AREA A TREATED AS LEVEL NUMBER.

An 01 level item was detected beyond Area A and accepted as if in Area A.

503 MULTIPLE PICTURES FOR SAME ITEM. LAST USED.

A data item has more than one PICTURE clause. The compiler used the last PICTURE clause specified.

504 CLOSING PARENTHESIS MISSING IN PICTURE.

The right parenthesis is missing in the PICTURE string. The compiler uses the last four characters of the PICTURE string.

505 NOT A SUBPROGRAM .PROGRAM. IGNORED.

An EXIT PROGRAM has been detected, but the COBOL program being compiled is not a subprogram. Because EXIT PROGRAM is meaningful only in a subprogram, the word PROGRAM is ignored, and the statement is treated as if it were a simple EXIT statement.

506 EXPANDED PICTURE STRING TOO LONG. PIC X ASSUMED.

The expansion of a PICTURE string specification produces a string that exceeds implementation limitations. The compiler ignores the user-supplied PICTURE and treats the data item as if it had a "PICTURE X" declaration.

507 SPECIFIER OMITTED BEFORE LEFT PAREN. IN PIC.

The first character of a PICTURE string is a left parenthesis. The compiler ignores the user-supplied PICTURE and treats the data item as alphanumeric with a "PICTURE X" declaration.

510 SECTION NO. GREATER THAN 49 TREATED AS 49.

A segment number greater than 49 follows the word SECTION. The segment is treated as if it were 49.

511 INVALID ITEM LENGTH IN PARENTHESES OF PICTURE.

The parenthesized length specifier in a PICTURE contains non-numeric characters. The compiler ignores the user-supplied PICTURE and treats the data item as alphanumeric with a "PICTURE X" declaration.

512 VALUE CLAUSE NOT ALLOWED IN LINKAGE SECTION.

The VALUE clause cannot appear in data items in the LINKAGE SECTION. The only place the VALUE clause can appear in the LINKAGE SECTION is in a condition name definition.

513 OPERAND IN .USING. MUST BE LINKAGE SECTION ITEM.

Only level 01 or 77 LINKAGE SECTION items may appear in the USING phrase of a PROCEDURE DIVISION header. Fatal.

514 MULTIPLE FLOATING FIELDS IN NUMERIC EDIT ITEM.

The PICTURE string contains multiple floating fields. The compiler ignores the user-supplied PICTURE and treats the data item as alphanumeric with a "PICTURE X" declaration.

515 MULTIPLE ZERO SUPPRESS FIELDS IN PICTURE STRING.

Multiple zero suppression fields are detected in a PICTURE string. The compiler ignores the user-supplied PICTURE and treats the data item as alphanumeric with a "PICTURE X" declaration.

516 ZERO SUPPRESSION ILLEGAL WITH FLOATING FIELD.

The PICTURE string contains both floating and zero suppression fields. The compiler ignores the user-supplied PICTURE and treats the data item as alphanumeric with a "PICTURE X" declaration.

517 ILLEGAL SYNTAX IN PICTURE STRING.

The PICTURE string is not specified correctly according to the rules of PICTURE string syntax. The compiler ignores the user-supplied PICTURE and treats the data item as alphanumeric with a "PICTURE X" declaration.

520 MULTIPLE DECIMAL POINTS IN PICTURE.

The PICTURE string contains multiple decimal point specifications (V's, P's, or periods). The compiler ignores the user-supplied PICTURE and treats the data item as alphanumeric with a "PICTURE X" declaration.

521 OPERAND IN USING MUST BE LEVEL 01 OR 77.

Only level 01 or 77 LINKAGE SECTION items may appear in the USING phrase of a PROCEDURE DIVISION header. Fatal.

522 INVALID USAGE. IGNORED.

The USAGE clause contains an invalid word. The compiler ignores the entire USAGE clause.

523 MULTIPLE USAGE CLAUSES. LAST USED.

The defined data-name has multiple USAGE clauses specified. The last USAGE clause specified is used by the compiler.

524 MULTIPLE OCCURS CLAUSES. LAST USED.

The defined data-name has multiple OCCURS clauses specified. The compiler uses the last OCCURS clause specified.

525 OCCURS SPECIFICATION ERROR. 1 ASSUMED.

The integer entry of the OCCURS clause is either non-numeric or non-integer or is not in the range 1 to 4095. The compiler assumes an integer value of 1.

526 DATANAME OMITTED IN DATA DESCRIPTION ENTRY.

The data-name declaration is omitted after a level-number in the data description entry. The compiler supplies a system-defined name and proceeds with the processing of the data description entry. The system-defined name is transparent and, thus, inaccessible to the user.

527 INVALID INDEX NAME. IGNORED.

The compiler did not recognize a valid index name in the INDEXED BY phrase. The compiler ignores the INDEXED BY phrase.

530 USAGE OPTION NOT YET IMPLEMENTED. IGNORED.

The compiler detected COMP-1 in the USAGE clause. This option is not implemented and is ignored. The default USAGE of DISPLAY is used by the compiler.

531 TERMINATOR OMITTED AFTER DATAITEM DESCRIPTION.

A data description entry in the DATA DIVISION is not terminated by a period. The compiler assumes the period is present and continues processing.

532 INVALID SIGN IN NUMERIC PICTURE.

The sign character S is detected in a position other than the leading character position of a numeric PICTURE string. The compiler ignores the user-supplied PICTURE and treats the data item as alphanumeric with a "PICTURE X" declaration.

533 PICTURE CLAUSE OMITTED ON ELEMENTARY ITEM.

An elementary item is recognized with its PICTURE clause omitted in the description. The compiler treats the data item as alphanumeric with a PICTURE X declaration.

534 NUMERIC ITEM EXCEEDS 18 DIGIT MAX. TRUNCATED.

A numeric field is defined in this PICTURE with more than 18 digits of precision. The numeric field is truncated to 18 digits.

535 COMP ITEM EXCEEDS 18 DIGITS. ASSIGN 4 WORDS.

A COMPUTATIONAL data item exceeds 18 digits in its specification. The compiler truncates it and allocates four words for its run-time storage.

536 INDEX ITEM HAS ILLEGAL CLAUSE.

The compiler recognized a JUSTIFIED, SYNCHRONIZED, VALUE, PICTURE, or SIGN clause on a data-item description that has INDEX USAGE. The compiler ignores the offensive clause.

537 NUMERIC VALUE FOR DISPLAY ITEM. IGNORED.

The VALUE clause specifies numeric value initialization for a non-numeric data-item that is defined with DISPLAY USAGE. The VALUE clause is ignored.

540 VALUE TOO LONG. TRUNCATED.

The non-numeric literal in the VALUE clause is longer than the associated data-item. The literal is truncated on the right to fit in the storage allocated to the data-item.

541 CLAUSE DUPLICATION. IGNORED.

This clause has been previously recognized for this item. The duplicate clause is ignored.

542 INVALID WORD IN .BLANK WHEN ZERO.. IGNORED.

The keyword ZERO was not recognized in the BLANK WHEN ZERO clause. The entire clause is ignored.

543 LEVEL NUMS UNEQUAL IN .REDEFINES. CLAUSE IGNORED.

A REDEFINES clause attempts to redefine two items of different level numbers. The REDEFINES clause is ignored.

544 POSSIBLE OVERLAP OF DEPENDING ON ITEM AND TABLE.

The DEPENDING ON item and variable length table are both defined in the LINKAGE SECTION. Because LINKAGE SECTION items are associated with data items appearing in a CALL statement, there is no way at compile time to ensure that the DEPENDING ON items and table do not overlap. The COBOL run-time RTS does not check for overlap of the DEPENDING ON item and the table during execution. It is, therefore, your responsibility to ensure that overlap does not occur.

545 LEVEL ILLEGAL AFTER 77. TREATED AS 01.

An invalid level number (02-49) follows a 77 level item. The 77 level item is treated as an 01 level item. This action can cause further diagnostics if it is not a valid group item.

546 PERIOD OMITTED AFTER .EXIT PROGRAM.

The words EXIT PROGRAM are not followed by a period. The error is ignored.

547 .EXIT PROGRAM. NOT LASTSTMT OF SENTENCE.

An EXIT PROGRAM statement appears in a sequence of statements within a sentence. But, it is not the last statement. All of the statements following it are compiled, but can never be executed.

550 REDEFINING LENGTH SHOULD MATCH ORIGINAL LENGTH.

The length of a non-01 level REDEFINES item is not the same as the length of the item it REDEFINES. The new length is used.

551 REDEFINITION OF .OCCURS. ITEM. IGNORED.

Items with OCCURS cannot be redefined. REDEFINES is ignored.

552 PROCESSING RESUMES AFTER BAD FD.

Prior to issuing this message, the compiler discovered bad syntax in the FD of the FILE SECTION. The compiler at that time issued an error message identifying the syntax error. Then the compiler attempted to recognize another FD, the WORKING-STORAGE SECTION header or the PROCEDURE DIVISION. Upon recognizing one of these three language elements, the compiler issues this diagnostic to indicate that normal processing has resumed.

553 INVALID CLAUSE KEYWORD. OTHER CLAUSES SKIPPED.

A reserved clause keyword was expected at this point in a data item description entry of the DATA DIVISION, but was not recognized by the compiler. The compiler skips to the next level number data item description.

554 INVALID WORD FOLLOWING .VALUE.. IGNORED.

The VALUE clause contains an invalid word for this data description. The entire VALUE clause is ignored.

555 VALUE CONFLICT. GROUP VALUE USED.

The VALUE clause assigns a value to an item subordinate to a group item that also has a VALUE clause. The subordinate VALUE clause is ignored.

556 LEVEL NUMBER OMITTED. ITEM IGNORED.

The level number has been omitted in a data-item description. All source text is ignored up to and including the next period.

557 NO VALUE AFTER CONDITION NAME. 88 IGNORED.

An 88 level condition-name has no VALUE clause specified. The entire 88 level data-item is ignored.

560 SYNTAX ERROR IN SWITCH CLAUSE. CLAUSE IGNORED.

The SWITCH clause has a syntax error in its specification. The compiler ignores the entire clause.

561 .NO. MISSING IN ADVANCING PHRASE. ASSUMED.

The keyword NO is missing in the ADVANCING phrase of the DISPLAY statement. NO is assumed present.

562 .ADVANCING. MISSING AFTER .NO.. ASSUMED.

The keyword ADVANCING is missing in the ADVANCING phrase of the DISPLAY statement. ADVANCING is assumed present.

563 DUPLICATE DATANAME DECLARATION DETECTED.

In the ENVIRONMENT DIVISION and/or DATA DIVISION, a data-name is defined that is not uniquely referenceable even with complete qualification.

564 ILLEGAL PARAGRAPH HEADER ID DIV. PAR IGNORED.

An illegal paragraph header appears in the IDENTIFICATION DIVISION. The paragraph is ignored.

565 ILLEGAL PARAGRAPH HEADER ENV DIV. PAR IGNORED.

An illegal paragraph header appears in the ENVIRONMENT DIVISION. The paragraph is ignored.

566 NUMERIC LITERAL ILLEGAL ON GROUP ITEM. IGNORED.

A numeric literal is illegal in the VALUE clause of a group item. The VALUE clause is ignored.

567 .ENVIRONMENT. NOT FOLLOWED BY .DIVISION..

The word ENVIRONMENT is not followed by the word DIVISION. DIVISION is assumed present.

570 TERMINATOR MISSING AFTER .DATA DIVISION. HEADER.

The DATA DIVISION header is not followed by a period. The period is assumed present and processing continues.

571 TERMINATOR MISSING AFTER PARAGRAPH HEADER.

A paragraph header in the IDENTIFICATION or ENVIRONMENT DIVISION is not terminated by a period. The period is assumed present and processing continues.

572 .RENAMES. SPECIFIES STORAGE OVERLAP ON RIGHT.

In processing the RENAMES clause, the compiler detects the condition in which the end of the storage allocated to the data-name after the THRU keyword is not to the right of the end of the storage allocated to the data-name after the RENAMES keyword. The compiler ignores the entire RENAMES data description entry.

573 .SECTION. OMITTED FROM SECTION HEADER.

An ENVIRONMENT DIVISION section name is not followed by the word SECTION. The error is ignored.

574 TERMINATOR MISSING AFTER SECTION HEADER.

An ENVIRONMENT DIVISION section header is not terminated by a period. The error is ignored.

600 ILLEGAL LEVEL NUMBER. TREAT AS 01.

This level number is not an 01-49, 66, 77, or 88 level number. The level number is assumed to be 01.

601 TERMINATOR MISSING AFTER ENV DIV HEADER.

The ENVIRONMENT DIVISION header is not terminated by a period. The period is assumed present and processing continues.

602 .DATA. NOT FOLLOWED BY .DIVISION.

The word DATA is not followed by the word DIVISION. DIVISION is assumed present.

603 ENVIRONMENT DIVISION HEADER OMITTED.

The program contains no ENVIRONMENT DIVISION header. The compiler resumes processing at the next paragraph header.

604 UNRECOGNIZABLE COBOL PROGRAM FORMAT. ABORT.

The compiler is unable to recognize the reserved word IDENTIFICATION as the first word required in a COBOL source program. Failure to recognize this required reserved word may be due to one of the following reasons:

(1) IDENTIFICATION is, in fact, omitted as the first word of the source file, (2) the user is attempting to compile a COBOL source program in conventional format without specifying the conventional format switch, or (3) the user is attempting to compile a file that is not a COBOL source program. The compiler issues a string of diagnostics and then aborts the compilation.

605 .IDENTIFICATION. NOT FOLLOWED BY .DIVISION..

The word IDENTIFICATION is not followed by the word DIVISION. DIVISION is assumed present.

606 TERMINATOR OMITTED AFTER .ID DIVISION. HEADER.

The IDENTIFICATION DIVISION header is not terminated by a period. The period is assumed present and processing continues.

607 .PROGRAMID. EXPECTED AFTER DIVISION HEADER.

The IDENTIFICATION DIVISION header is not followed by the PROGRAM-ID paragraph. The error is ignored and processing continues.

610 TERMINATOR OMITTED AFTER .PROGID. PARA HEADER.

The PROGRAM-ID paragraph-name is not terminated by a period. The period is assumed present and processing continues.

611 INVALID PROGRAM NAME IN .PROGRAM ID. PARAGRAPH.

The program name of the PROGRAM-ID paragraph contains an invalid character or exceeds the maximum length. The error is ignored and processing continues.

612 TOO MANY FILES FOR LUNS OR TEMPORARY SPACE.

The compiler has discovered either that more than 30 files are declared in the program or that more than 30 SAME RECORD AREA clauses are specified in the program. The compiler imposes a limit of 30 in both cases, because the associated compiler and/or run time table space is exhausted.

613 INVALID WORD SUSPENDS PROCESSING. SCAN FORWARD.

An unidentifiable word is found where a verb is expected. The compiler scans to a verb, period, or word in Area A.

614 PROCESSING RESTARTS ON VERB.

Due to a previous syntax error, the compiler scanned forward for the next verb, period, or Area A word at which to resume compilation. The compiler recognized a verb and resumes normal compilation at this point. This message is an observation only.

615 PROCESSING RESTARTS ON PROCEDURE NAME.

Due to a previous syntax error, the compiler scanned forward for the next verb, period, or Area A word at which to resume compilation. The compiler recognized an Area A word and resumes compilation at this point. This message is an observation only.

616 PROCESSING RESTARTS AFTER TERMINATOR.

Due to a previous syntax error, the compiler scanned forward for the next verb, period, or Area A word at which to resume compilation. The compiler recognized a period and resumes normal compilation on the word following the period. This is an observation only.

617 .IDENTIFICATION. KEYWORD NOT IN AREA A.

The compiler detects that the IDENTIFICATION keyword is not in Area A. The compiler ignores the error and continues processing.

620 PARAGRAPH TERMINATOR ASSUMED OMITTED.

A paragraph was terminated without a period. The period is assumed and processing continues.

621 .LINAGE. INVALID FOR THIS FILE. CLAUSE IGNORED.

The LINAGE clause must not be specified for a file that has RELATIVE or INDEXED organization. The LINAGE clause is ignored.

622 TERMINATOR MISSING AFTER PROCEDURE NAME.

A section or paragraph name is not terminated by a period. The period is assumed present and processing continues.

623 .ELSE DOES NOT HAVE ASSOCIATED .IF.. IGNORED.

The word ELSE has no associated IF statement. The ELSE is ignored.

624 VERB EXPECTED TO FOLLOW ELSE.. .ELSE. IGNORED.

A sentence ends with the word ELSE. The ELSE is ignored.

625 .JUSTIFY. WITH NUMERIC OR EDITED ITEM. IGNORED.

The JUSTIFIED clause must not be specified for a numeric or numeric-edited data item. The JUSTIFIED clause is ignored.

626 .BLANK WHEN ZERO. ILLEGALLY SPECIFIED. IGNORED.

The BLANK WHEN ZERO clause must be specified only for a numeric or numeric-edited data item. The clause is ignored.

627 INVALID OR MISSING DATANAME AFTER .REDEFINES..

The compiler detects the omission of a valid data-name reference following the keyword REDEFINES. The compiler ignores the REDEFINES clause and continues processing the data description entry.

630 .REDEFINES. MUST FOLLOW DATA NAME. IGNORED.

The REDEFINES keyword appears in the wrong position of a data description entry. The REDEFINES clause is ignored.

631 DEPTH OF NESTED .IF. EXCEEDS LIMIT.

A nested IF statement has exceeded the maximum depth of 30 levels. The compiler ignores nesting beyond this depth.

632 DUPLICATE PROCEDURE NAME DETECTED.

In the Procedure Division, a paragraph or section-name is defined that is not uniquely referenceable even with qualification.

633 REFERENCE TO UNDEFINED PARAGRAPH NAME.

In the Procedure Division, an explicit qualified reference is made to a paragraph-name that is undefined in the section specified by the qualifier.

634 FILENAME LITERAL TOO LONG. TRUNCATED.

A file specification in the ASSIGN clause exceeds 150 characters in length. It is truncated to 150 characters.

635 ILLEGAL SYNTAX IN .GO TO. STATEMENT.

The compiler detects illegal syntax in the GO TO statement. Fatal.

636 INVALID INTEGER OR DATANAME.

In the LINAGE clause, the compiler failed to recognize a non-negative integer literal or a numeric integer data-name. This phrase of the LINAGE clause is ignored.

637 .GO TO. HAS MULTIPLE PROCEDURE NAMES.

A GO TO statement without the DEPENDING ON phrase has more than one procedure-name. Fatal.

640 INVALID WORD FOLLOWS .DATA DIVISION.

The word following the DATA DIVISION header either does not start in Area A or is not one of the reserved words FILE, WORKING-STORAGE, LINKAGE, or PROCEDURE. The compiler skips all source text until one of the keywords FILE, WORKING-STORAGE, LINKAGE, or PROCEDURE is recognized.

641 INVALID WORD IN FILE SECTION. SCAN FORWARD.

An invalid word was detected in the FILE SECTION where the keyword FD is expected. The compiler skips all source text until one of the keywords FD, WORKING-STORAGE, LINKAGE, or PROCEDURE is recognized.

642 .OMITTED LABELS IGNORED WITH .VALUE OF ID.

The LABEL RECORDS ARE OMITTED clause is ignored if VALUE OF ID is specified for a file. STANDARD labels are assumed. Warning.

643 .SECTION. EXPECTED AFTER HEADER WORD.

The keyword SECTION is omitted after the word FILE, WORKING-STORAGE, OR LINKAGE SECTION. It is assumed present and processing continues.

644 TERMINATOR EXPECTED AFTER SECTION HEADER.

The FILE SECTION, WORKING-STORAGE SECTION, or LINKAGE SECTION header is not terminated by a period. The period is assumed and processing continues.

646 .OF. OR .ID. MISSING IN .VALUE OF ID..

One or both of the keywords OF or ID is omitted in the VALUE OF ID clause. Their presence is assumed and processing continues.

647 ILLEGAL WORD IN AREA A. SCAN FORWARD.

In the WORKING-STORAGE SECTION, an 01 or 77 level number or the PROCEDURE keyword was expected in Area A, but was not recognized. The compiler skips all source text until one of the three expected language elements is recognized in Area A.

650 GROUP LEVEL .VALUE. DISALLOWED.

The VALUE clause on this group item is not permitted because a subordinate elementary item has a non-DISPLAY usage specified or has a SYNCHRONIZED clause specified. The group VALUE clause is ignored.

551 REFERENCED LINKAGE SECTION ITEM NOT ID .PD. USING..

This LINKAGE SECTION item has been referenced in the PROCEDURE DIVISION. However, neither this item nor the level 01 to which it is subordinate appeared in the PROCEDURE DIVISION USING phrase. Only those LINKAGE SECTION items appearing in the PROCEDURE DIVISION USING phrase, or items subordinate to them, may be referenced in the PROCEDURE DIVISION of a COBOL program. Fatal.

652 NON-SEQ FILE IN .MULTIPLE. FILE TAPE. CLAUSE.

In the I-O CONTROL paragraph, the MULTIPLE FILE TAPE clause is specified for a file whose organization is not SEQUENTIAL. The MULTIPLE FILE TAPE clause is ignored for this file.

653 .VALUE. CLAUSE ILLEGAL IN FILE SECTION.

A VALUE clause is specified for a data description entry given in the FILE SECTION. The VALUE clause is ignored.

654 SYNTAX ERROR IN CURRENCY CLAUSE.

The alphanumeric literal expected in the CURRENCY SIGN clause of the SPECIAL-NAMES paragraph is omitted. The clause is ignored and the currency sign defaults to the dollar sign.

655 ILLEGAL CURRENCY SIGN.

The alphanumeric literal in the CURRENCY SIGN clause is not allowed as the currency sign either because the literal is longer than one character or because it is an invalid COBOL currency sign. The CURRENCY SIGN clause is ignored, and the currency sign defaults to the dollar sign.

656 SPECIALNAMES CLAUSE INVALID.

An unrecognizable word appears in a position where a SPECIAL-NAMES paragraph clause keyword is expected. All source text is skipped until the next keyword is recognized.

657 SYNTAX ERROR IN DECIMALPOINT CLAUSE.

The keyword COMMA is omitted in the DECIMAL-POINT IS COMMA clause of the SPECIAL-NAMES paragraph. The clause is ignored.

660 .AFTER. MISSING IN .USE. STATEMENT. ASSUMED.

The keyword AFTER is omitted in the USE statement. AFTER is assumed present and processing continues.

661 NO .ERROR. OR .EXCEPTION. IN .USE. ASSUMED.

One of the keywords ERROR or EXCEPTION is omitted in the USE statement. The missing keyword is assumed present and processing continues.

662 NO KNOWN CLAUSES IN SPECIALNAMES.

The SPECIAL-NAMES paragraph contains no valid clauses. This is an observation only.

663 REDUNDANT .USE. COVERAGE. PREV. .USE. IGNORED.

Multiple USE statements have referenced the same file. The last USE statement specified is then applied to the referenced file. Fatal.

664 UNKNOWN OPEN MODE IN .USE. STATEMENT.

An unrecognizable OPEN mode option was specified in the USE statement. Fatal.

665 GROUP ITEM HAS BEEN CALLED FILLER.

A FILLER item cannot have any elementary items subordinate to it. The compiler replaces the FILLER declaration with a system-defined name and proceeds with the processing of the newly-named group item. The system-defined name is transparent and inaccessible to the user.

666 MISSING ENVIRONMENT DIVISION.

The program does not contain an ENVIRONMENT DIVISION. The compiler skips to the DATA DIVISION and continues processing.

667 DIVISION BY ZERO.

The divisor of a DIVIDE statement is a literal of zero value. The error is ignored.

670 VALUE NOT PERMITTED WITH THIS ITEM.

A VALUE clause is recognized in a data description entry that contains a REDEFINES or an OCCURS clause. The VALUE clause is ignored.

671 INVALID CONSTANT OR LITERAL FOLLOWING .ALL..

The reserved word ALL is not followed by a non-numeric literal or a figurative constant. ALL is ignored and processing continues.

672 BAD FILENAME IN .USE. STATEMENT.

An unrecognizable word appears where a file-name is expected in the USE statement. Fatal.

673 FILE NOT CLOSED.

The referenced file was opened, but there was no CLOSE statement detected for this file in the program.

674 SUBJECT OF .ALTER. IS SECTION NAME.

The ALTER statement references a section name. Only paragraph names may be altered. If this statement is reached during execution, the program will be aborted.

675 FILE COVERED BY CONFLICTING USE PROCEDURE.

There was more than one conflicting USE procedure specified for the referenced file. Fatal.

676 DATA DIVISION EXCEEDS ADDRESS RANGE.

The maximum DATA DIVISION size is 65,535 bytes. Fatal.

677 SUPPLIED VALUE INVALID FOR NUM ITEM. IGNORED.

The VALUE clause specifies invalid value initialization for a numeric data item. The compiler ignores the VALUE clause.

700 FILE ACCESSED BY VERB REQUIRING REL. OR IDX ORG.

A file whose organization is SEQUENTIAL is referenced by the START or DELETE verbs or by an I/O verb that has the INVALID KEY clause specified. In all these cases, the referenced file must have RELATIVE or INDEXED organization. Fatal.

701 FILE ACCESSED BY VERB REQ. SEQUENTIAL ORG.

A file whose organization is RELATIVE or INDEXED is referenced by an I/O verb that has the AT EOP or ADVANCING clauses specified. The referenced file must have SEQUENTIAL organization. Fatal.

702 VERB NOT IMPLEMENTED.

An ANS 1974 COBOL verb appears that is not implemented in this release of the compiler. The compiler scans to another verb, period, or word in Area A.

704 OCCURS ILLEGAL FOR 01 OR 77 ITEM. IGNORE.

An OCCURS clause is specified for an 01 or 77 level data-name. The compiler ignores the OCCURS clause.

705 .ACCEPT FROM. OBJECT NOT IN SPECIALNAMES.

The mnemonic-name used in the ACCEPT statement was not defined in the SPECIAL-NAMES paragraph. Fatal.

706 ACCEPT IDENTIFIER INVALID.

The word following the ACCEPT verb is not a data-name or is a data-name that has non-DISPLAY usage or invalid class. Fatal.

707 VERB OR COND. CLAUSE CONFLICTS WITH FILE ACCESS.

There is a conflict between the ACCESS MODE of the referenced file and the I/O verbs and/or condition clauses that reference this file. Fatal.

710 DATANAME AFTER .GO DEPENDING. INVALID.

The word following the DEPENDING ON phrase of the GO TO statement is not a data-name or is a data-name that has INDEX usage. Fatal.

711 INVALID CLASS OF DATANAME AFTER .GO DEPENDING.

The data-name following the DEPENDING ON phrase of the GO TO statement is not a numeric data-name or is a numeric, non-integer data-name. Fatal.

712 .DISPLAY UPON. OBJECT NOT IN SPECIALNAMES.

The mnemonic-name used in the DISPLAY statement was not defined in the SPECIAL-NAMES paragraph. Fatal.

713 .DISPLAY. OPERAND IS INVALID.

A data item in the DISPLAY statement has invalid class or USAGE.

714 MISSING OR INVALID OPERAND FOR ARITHMETIC VERB.

One of the operands of an arithmetic statement is either missing or invalid. Fatal.

715 MISSING OR INVALID SOURCE OPERAND.

The source operand is missing following an arithmetic verb. Fatal.

716 MISSING OR INVALID DESTINATION OPERAND.

717 .GIVING. REQUIRED AFTER .DIV...BY.

The GIVING phrase INTO is missing in a DIVIDE...BY statement. Fatal.

720 .GIVING. REQUIRED AFTER LITERAL OPERAND.

The GIVING phrase is required if the second operand of an ADD, DIVIDE, MULTIPLY, or SUBTRACT statement is a literal. Fatal.

721 .BY. MISSING IN .MULTIPLY.

The keyword BY is missing in a MULTIPLY statement. Fatal.

722 .BY. OR .INTO. MISSING FROM .DIVIDE.

One of the keywords BY or INTO is missing from the DIVIDE statement. Fatal.

723 .FROM. MISSING IN .SUBTRACT.

The keyword FROM is missing from the SUBTRACT statement. Fatal.

724 FILE NEEDS DYNAMIC ACCESS FOR .READ NEXT..

In a READ NEXT statement, the referenced file must have ACCESS MODE IS DYNAMIC specified in the FILE-CONTROL paragraph. Fatal.

725 BAD PROCEDURE NAME IN .PERFORM..

A missing or invalid procedure name is recognized in the PERFORM statement. Fatal.

726 ILLEGAL OPERAND OF .TIMES. OPTION OF .PERFORM..

The TIMES operand of the PERFORM statement is not a numeric integer data-name or numeric integer literal. The compiler assumes a value of 1 for the TIMES operand.

727 .TIMES. MISSING FROM .PERFORM.. ASSUMED.

The PERFORM statement does not contain the keyword TIMES but does contain the iteration value required to execute the PERFORM correctly. The keyword TIMES is assumed present.

730 PROCEDURE NAME OMITTED IN .ALTER..

A valid procedure-name was not recognized in the ALTER statement. Fatal.

731 ILLEGAL .ALTER. DUE TO MISSING .TO..

The keyword TO was not recognized in the ALTER statement. Fatal.

732 FILE HAS VAR. SIZE RECS. .READ INTO. ILLEGAL.

It is illegal for the READ INTO statement to reference a file that has multiple record descriptions of different lengths. Fatal.

733 FILE ACCESSED BY VERB REQUIRING .LINAGE.

A file that did not have a LINAGE clause in its specification is accessed by an I/O verb. Fatal.

734 .DELETE. OR .REWRITE. WITHOUT INV. KEY OR USE.

A DELETE or REWRITE statement without the INVALID KEY phrase references a file for which there is no USE procedure. Fatal.

735 OPEN MODE OR NO READ PROHIBITS REWRITE OR DELETE.

A DELETE or REWRITE statement references a file that was not OPENed in the proper mode or that has no READ statement referencing it in the program. Fatal.

736 .START. CONFLICTS WITH OPEN MODE.

A START statement references a file that was not opened in the proper mode. Fatal.

737 .WRITE. CONFLICTS WITH OPEN MODE.

A WRITE statement references a file that was not opened in the proper mode. Fatal.

740 .READ. CONFLICTS WITH OPEN MODE.

A READ statement references a file that is only opened in OUTPUT or EXTEND mode. Fatal.

741 USE NOT IN DECLAR. OR NOT FOLLOWING SECTION NAME.

The USE statement is not in the DECLARATIVES section of the PROCEDURE DIVISION or is not immediately following a section name inside the DECLARATIVES. Fatal.

742 MORE THAN 255 ALTERNATE KEYS. IGNORED.

The maximum of 255 ALTERNATE KEYS has been exceeded. The clause is ignored.

743 INTEGER IN SWITCH CLAUSE INVALID OR OMITTED.

A SWITCH clause of the SPECIAL-NAMES paragraph either contains an invalid numeric integer or has omitted the integer in its specification. A SWITCH clause integer must be in the decimal range 1<=n<=16. The SWITCH clause is ignored.

744 .IS. OMITTED IN SPECIALNAMES. ASSUMED PRESENT.

The required keyword IS is omitted in a clause of the SPECIAL-NAMES paragraph. IS is assumed present and processing continues.

745 DEVICE MNEMONIC OMITTED IN SPECIALNAMES.

A valid device mnemonic-name is not recognized in one of the CONSOLE, LINE-PRINTER, CARD-READER, PAPER-TAPE-READER, or PAPER-TAPE-PUNCH clauses of the SPECIAL-NAMES paragraph. All source text is skipped until the next recognizable keyword.

746 TERMINATOR OMITTED IN SPECIALNAMES.

The SPECIAL-NAMES paragraph is not terminated by a period. The period is assumed present and processing continues.

747 SUBJECT OF .ALTER. NOT .GO TO.. ALTER IGNORED.

The paragraph referenced by an ALTER statement does not contain a GO TO statement as its first statement. The ALTER statement is ignored.

750 KEYWORD OMITTED IN .SWITCH. CLAUSE.

One of the keywords OFF or ON is omitted in the SWITCH clause of the SPECIAL-NAMES paragraph. The SWITCH clause is ignored.

751 CONDITION NAME MISSING IN .SWITCH. CLAUSE.

A valid condition-name is not recognized in the SWITCH clause of the SPECIAL-NAMES paragraph. The SWITCH clause is ignored.

752 .CR. OR .DB. NOT AT RIGHT END OF PICTURE.

The PICTURE symbol CR or DB does not appear at the right end of the PICTURE string. The compiler ignores the user-supplied PICTURE and treats the data item as alphanumeric with a "PICTURE X" DECLARATION.

753 .CR. OR .DB. USED WITH SIGNED ITEM.

Both the PICTURE symbols, CR or DB, and a sign, + or -, appear in the same PICTURE. The compiler ignores the user-supplied PICTURE and treats the data item as alphanumeric with a "PICTURE X" declaration.

754 MULTIPLE DEFINITION OF SWITCH. FIRST USED.

Multiple definitions of a COBOL switch are detected in the SPECIAL-NAMES paragraph. All but the first definition of SWITCH are ignored.

755 .SENTENCE. ASSUMED AFTER .NEXT.

The keyword NEXT is not followed by the keyword SENTENCE. SENTENCE is assumed present and processing continues.

756 SUBSCRIPT NOT NUMERIC INTEGER.

A data-name used as a subscript is not numeric in class. A default value of 1 is assumed as the subscript.

760 ILLEGAL SYNTAX IN .DIVIDE. STATEMENT.

The compiler detects illegal syntax in the DIVIDE statement. Fatal.

761 INDEXED FILE REQUIRES .RECORD KEY. PHRASE.

Self explanatory.

762 RECORD KEY INVALID FOR THIS FILE.

The RECORD KEY clause is valid only for indexed files.

763 .ALT RECORD KEY. INVALID FOR FILE. IGNORED.

The ALTERNATE RECORD KEY clause is valid only for indexed files.

764 READ-AHEAD. OR. WRITE-BEHIND, NOT SUPPORTED.

The APPLY READ-AHEAD and APPLY WRITE-BEHIND clauses are not supported in this version of the compiler. The APPLY clause is ignored.

765 INTEGER INVALID IN. RESERVE AREA. CLAUSE.

The number of buffer areas reserved by the RESERVE clause is invalid. The clause is ignored, and the RMS default is used.

766 BAD VALUE IN BLOCK CONTAINS CLAUSE.

The numeric literal in the BLOCK clause is less than the sum of the record size, the record header size, and the bucket header size. The BLOCK CONTAINS clause is ignored.

767 VALUE IN. BLOCK CONTAINS. CLAUSE IS ROUNDED UP.

The numeric literal in the BLOCK clause is not a multiple of 512. The value is rounded up to the next even multiple of 512.

770 EXPECTED .RECORD KEY. DATANAME NOT DEFINED.

The data-name in a RECORD KEY clause has not been defined in the DATA DIVISION.

771 .RECORD KEY. DATANAME HAS INVALID CLASS.

A data-name referenced in a RECORD KEY or ALTERNATE RECORD KEY phrase of a SELECT clause in the FILE-CONTROL paragraph is defined with non-alphanumeric class in the FILE SECTION.

772 .RECORD KEY. DATA ITEM CANNOT BE VARIABLE LENGTH.

A data-name referenced in a RECORD KEY or ALTERNATE RECORD KEY phrase of a SELECT clause in the FILE-CONTROL paragraph is defined in the FILE SECTION as an item whose size is variable.

773 .RECORD KEY. ITEM NOT DEFINED IN RECORD OF FILE.

A data-name referenced in a RECORD KEY or an ALTERNATE RECORD KEY phrase of a SELECT clause is not defined in the record description of the associated file.

774 FILE ACCESSED BY VERB REQUIRING INDEXED ORG.

A file whose organization is SEQUENTIAL or RELATIVE is referenced by the READ verb that has the KEY IS data-name phrase specified. The referenced file must have INDEXED organization. Fatal.

775 .KEY IS. PHRASE INVALID FOR SEQUENTIAL .READ.

Either the file has ACCESS SEQUENTIAL or the READ statement contains the word NEXT. In either case the KEY IS data-name phrase is illegal. Fatal.

776 INVALID DATANAME IN .KEY IS. PHRASE.

The KEY IS phrase of the READ statement was not followed by a data-name. Fatal.

777 .KEY IS. PHRASE NOT FOLLOWED BY RECORD KEY.

The data-name following the KEY IS phrase of the READ statement is not a RECORD KEY or ALTERNATE RECORD KEY for the referenced file. The RECORD KEY data-name is assumed.

1000 VARIABLE OCCURRENCES TABLE MUST END RECORD.

A COBOL table declared with the DEPENDING ON phrase can be followed in the record only by data description entries whose level-numbers are greater than the level-number of this table entry. The compiler ignores the remainder of the record descriptor from the point where the error is detected. Fatal.

1001 .ASCENDING. OR .DESCENDING. DATANAME EXPECTED.

A user-defined data-name was expected, but not found, in the ASCENDING KEY IS or DESCENDING KEY IS phrase.

1002 RENAMED DATAITEMS NOT IN CURRENT RECORD.

The data items specified after the RENAMES keyword (that is, the data items being renamed) are defined outside of the current record description. The compiler ignores the entire RENAMES data description entry.

1003 MAXIMUM OCCURRENCES NOT GREATER THAN MINIMUM.

In a variable occurrence table declaration, the integer following the keyword TO (that is, the maximum) must be greater than the integer following the keyword OCCURS (that is, the minimum). The compiler assumes the maximum value to be one greater than the minimum value.

1004 .DEPENDING. IS OMITTED IN THE .OCCURS. CLAUSE.

In a variable occurrence table declaration, the keyword DEPENDING has been omitted. The compiler ignores the remainder of the OCCURS clause and treats the table declaration as an ordinary COBOL table.

1005 A DATANAME MUST FOLLOW THE .DEPENDING. KEYWORD.

In a variable occurrence table declaration, a valid data-name is not found following the keyword DEPENDING. The compiler ignores the remainder of the OCCURS clause and treats the table declaration as an ordinary COBOL table.

1006 .OCCURS DEPENDING. SUBORDINATE TO AN .OCCURS.

The compiler detects a table declaration with a DEPENDING ON phrase subordinate to a group item that has an OCCURS clause. The compiler ignores the DEPENDING ON phrase and treats the declaration as an ordinary COBOL table.

1007 MAXIMUM NO. TABLE OCCURRENCES MUST BE POSITIVE.

In a variable occurrence table declaration, the integer following the keyword TO (that is, the maximum) must be greater than zero. The compiler assumes the maximum value to be one greater than the integer value following the keyword OCCURS (that is, the minimum).

1010 EXPECTED .DEPENDING ON. DATANAME NOT DEFINED.

The data-name referenced in a DEPENDING ON phrase was not defined in the DATA DIVISION. Fatal.

1011 EXPECTED .ASCENDING KEY. DATANAME NOT DEFINED.

The data-name referenced in an ASCENDING KEY phrase was not defined in the DATA DIVISION. Fatal.

1012 EXPECTED .DESCENDING KEY. DATANAME NOT DEFINED.

The data-name referenced in a DESCENDING KEY phrase was not defined in the DATA DIVISION. Fatal.

1013 .DEPENDING ON. DATANAME NOT A NUMERIC INTEGER.

The data-name referenced in a DEPENDING ON phrase was not declared as a numeric integer in the DATA DIVISION. Fatal.

1014 .RENAMES. APPLIED TO AN INVALID LEVEL OF DATA.

The RENAMES clause specifies the renaming of data items whose level number is 01, 66, 77, or 88. The compiler ignores the entire RENAMES data description entry.

1015 .DEPENDING ON. DATANAME DETECTED WITHIN TABLE.

The compiler detects a data-name, that follows a DEPENDING ON phrase and that defines the current number of occurrences in a variable occurrence table, to have its storage allocated within the range of the table. Fatal.

1016 .OCCURS. CLAUSE ON A TABLE KEY DATANAME.

The compiler detects the presence of an OCCURS clause on a data item that has been declared as an ASCENDING or DESCENDING KEY. Fatal.

1017 .SEARCH ALL. TABLE DOES NOT HAVE KEYS.

The table being searched by a SEARCH ALL statement must have the ASCENDING KEY or DESCENDING KEY phrase specified in its declaration. Fatal.

1020 IMPERATIVE STATEMENT EXPECTED DURING .SEARCH.

A period or a non-imperative statement was found where the SEARCH statement environment is expecting an imperative statement. Fatal.

1021 KEYS SPECIFIED FOR .SEARCH ALL. NOT DENSE.

When a key is referenced for the SEARCH ALL statement, all preceding keys in the KEY clause of the table declaration must also be referenced. Fatal.

1022 .WHEN. EXPECTED BUT NOT FOUND IN .SEARCH.

The compiler expected but failed to recognize the WHEN keyword while compiling the SEARCH statement. Fatal.

1023 THE KEYWORD .WHEN. ILLEGAL IN THIS CONTEXT.

The compiler detects the presence of the keyword WHEN outside the environment of the SEARCH statement. Fatal.

1024 THE KEYWORD .SEARCH. ILLEGAL IN THIS CONTEXT.

While compiling a SEARCH statement, the compiler detects the presence of another SEARCH statement. The second SEARCH statement is detected at a point where an imperative statement is expected. Fatal.

1025 KEY MUST BE SUBSCRIPTED BY FIRST INDEX OF TABLE.

The SEARCH ALL statement requires that the key referenced on the left side of the simple condition must be subscripted by the first index name of the table being searched. Fatal.

1026 THE KEYWORD .SENTENCE. EXPECTED AFTER .NEXT..

The keyword SENTENCE was not detected after the NEXT keyword during the compilation of a SEARCH statement. Fatal.

1027 TABLE NAME NOT FOUND AFTER .SEARCH. VERB.

The compiler failed to recognize a valid table data item after the keyword SEARCH or SEARCH ALL. Fatal.

1030 INVALID TABLE REFERENCE IN .SEARCH. STATEMENT.

The table data item reference following the SEARCH or SEARCH ALL verbs must have both the INDEXED BY and the OCCURS clauses specified in its declaration. Fatal.

1031 DATANAME EXPECTED AFTER .VARYING. IN .SEARCH.

No data-name reference was found after the VARYING keyword in the SEARCH statement being compiled. Fatal.

1032 .VARYING. ITEM MUST BE INDEX OR INTEGER.

The data-name reference following the VARYING keyword must be an index data item, an index-name, or an elementary, numeric, integer data-name reference. Fatal.

1033 .SEARCH ALL. DATA ITEM IS NOT A KEY.

The data item referenced on the left side of the SEARCH ALL simple condition must be declared as an ASCENDING or DESCENDING KEY. Fatal.

1034 DATA ITEM NOT A KEY FOR THIS .SEARCH. TABLE.

The data item referenced on the left side of the SEARCH ALL simple condition is not a key for the table being searched. Fatal.

1035 .RENAMES. SPECIFIES RENAMING OF A COBOL TABLE.

The RENAMES clause specifies the renaming of an item that has an OCCURS clause in its declaration or is subordinate to another item having an OCCURS clause. The compiler ignores the entire RENAMES data description entry.

1036 .RENAMES. APPLIED TO VARIABLE LENGTH DATAITEM.

The compiler detects an application of the RENAMES clause to a range of data items that contains a data item whose length is variable at run-time because is has a subordinate data item whose data description entry contains an OCCURS DEPENDING ON clause. The compiler ignores the entire RENAMES data description entry.

1037 DATANAME OMITTED AFTER 66 LEVEL NUMBER.

The data-name declaration is omitted after a 66 level number. The compiler ignores the entire RENAMES data description entry.

1040 .RENAMES. OMITTED IN LEVEL 66 DESCRIPTION ENTRY.

The RENAMES keyword is omitted in a level 66 data description entry. The compiler ignores the entire level 66 data description entry.

1041 SEARCH KEY NOT SUBORDINATE TO TABLE.

The compiler detects an ASCENDING or DESCENDING data-name key that is not defined as a data item subordinate to the associated SEARCH table.

1042 INVALID OR MISSING DATANAME AFTER .RENAMES..

The data-name is missing after the RENAMES keyword or, if present, is not recognized as a valid data item that was previously defined. The compiler ignores the entire RENAMES data description entry.

1043 .OCCURS. ITEM NOT ALLOWED BETWEEN TABLE AND KEY.

The compiler detects a data item declared with an OCCURS clause "sandwiched" between the declaration of another COBOL table and its associated SEARCH key.

1044 .RENAMES. SPECIFIES INVALID NOMENCLATURE RANGE.

In processing the RENAMES clause, the compiler detects an invalid nomenclature range specified by identical data-names following the RENAMES and THRU keywords, respectively. The compiler ignores the entire RENAMES data description entry.

1045 .RENAMES. SPECIFIES STORAGE OVERLAP ON LEFT END.

In processing the RENAMES clause, the compiler detects the condition in which the beginning of the storage allocated to the data-name after the THRU keyword is to the left of the beginning of the storage allocated to the data-name after the RENAMES keyword. The compiler ignores the entire RENAMES data description entry.

1046 INVALID OR MISSING DATANAME AFTER .THRU..

In specifying the RENAMES clause, a data-name is missing after the THRU keyword or, if present, is not recognized as a valid data item that was previously defined. The compiler ignores the entire RENAMES data description entry.

1047 DATANAME MISSING AFTER .CORRESPONDING.

In the processing of an ADD, SUBTRACT, or MOVE CORRESPONDING statement, the compiler detects the omission of a valid data-name reference after the CORRESPONDING keyword. Fatal.

1050 .TO. OR .FROM. OMITTED IN .CORRESPONDING..

In the processing of an ADD, SUBTRACT, or MOVE CORRESPONDING statement, the compiler detects the omission of the TO or FROM keyword. Fatal.

1051 INVALID OR MISSING DATANAME AFTER .TO. OR .FROM.

In the processing of an ADD, SUBTRACT, or MOVE CORRESPONDING statement, the compiler detects the omission of a valid data-name reference after the keyword TO or FROM. Fatal.

1052 NO OBJECT CODE PRODUCED FOR .CORRESPONDING.

In the processing of an ADD, SUBTRACT, or MOVE CORRESPONDING statement, the compiler produced no object code because no "correspondence" was found between the two group items referenced in the COBOL statement containing the CORRESPONDING option. This diagnostic is informational only.

1053 GROUP ITEM NOT REFERENCED IN .CORRESPONDING.

In the processing of an ADD, SUBTRACT, or MOVE CORRESPONDING statement, the compiler discovered that one of the references is a reference to an elementary item. Fatal.

1054 LEVEL 66 REFERENCE DISALLOWED IN .CORRESPONDING.

In the processing of an ADD, SUBTRACT, or MOVE CORRESPONDING statement, the compiler detects a reference to a

data-name declared at level 66. This is an invalid reference. Fatal.

1055 .FILE STATUS. ITEM DEFINED IN .FILE SECTION.

A data-name referenced in a FILE STATUS phrase of a SELECT clause is defined in the FILE SECTION of the COBOL program. The compiler ignores this error and continues to process the FILE STATUS data-name.

1056 INCOMPATIBLE OPERANDS FOUND IN .CORRESPONDING.

In the processing of an ADD, SUBTRACT, or MOVE CORRESPONDING statement, the compiler detects a pair of CORRESPONDING data items that are incompatible. This diagnostic is informational only.

1057 EMPTY .GO TO. WAS NOT THE SUBJECT OF AN .ALTER..

A GO TO statement without a procedure reference was detected. The empty GO TO is not the subject of an ALTER statement. Fatal.

1060 QUALIFIER OMITTED IN PROCEDURE REFERENCE.

A section name is omitted after the keyword OF or IN in a qualified procedure reference of the COBOL statement being compiled. Fatal.

1061 INCONSISTENT NUMBER OF ARGUMENTS IN .CALL..

The subprogram referenced in this CALL statement has been referenced before. The number of arguments in the earlier CALL differs from the number in the current CALL.

1062 PARAGRAPH WITHOUT SECTION PRECEDES THIS SECTION.

In a COBOL program, if one paragraph is in a section, then all paragraphs must be in sections. In this source program, a paragraph not within a section has been detected preceding this section in the source program.

1063 DUPLICATE PARAGRAPH NAME DETECTED.

In a section of the Procedure Division, a paragraph name is defined more than once and is not uniquely referenceable even with qualification.

1064 REFERENCE TO UNDEFINED PROCEDURE NAME,

The compiler detects a reference to an undefined procedure name in the PROCEDURE DIVISION.

1065 UNDEFINED PROCEDURE QUALIFIER REFERENCE.

The compiler detects a qualified procedure reference that contains an undefined qualifier in the PROCEDURE DIVISION.

1066 ILLEGAL PROCEDURE NAME REFERENCE.

The compiler detects an invalid procedure name reference in the PROCEDURE DIVISION.

1067 AMBIGUOUS PROCEDURE NAME REFERENCE.

The compiler detects a reference in the PROCEDURE DIVISION to a procedure name that is not uniquely referenceable, even through qualification.

1070 PARAGRAPH NAME DISALLOWED AS QUALIFIER.

The compiler detects a qualified procedure reference in which the qualifier is a paragraph name.

1071 SECTION NAME REFERENCE MAY NOT BE QUALIFIED.

The compiler detects a qualified procedure reference in which a section name is qualified by another section name.

1072 AMBIGUOUS PARAGRAPH NAME REFERENCE.

The compiler detects a reference in the PROCEDURE DIVISION to a paragraph name that is not uniquely referenceable, even through qualification.

1073 POSSIBLE .PERFORM. RANGE VIOLATION.

The compiler detects a PERFORM THRU statement in which the procedure name following THRU is defined before the procedure name following the PERFORM. This condition could a logic problem in the COBOL program being compiled.

1074 NUMERIC PROCEDURE NAME EXCEEDS 30 CHARACTERS.

A numeric string that appears to be a numeric procedure name exceeds 30 characters in length. The string is truncated on the right to 30 characters and processing of the numeric procedure name continues.

1075 NUMERIC PROCEDURE NAME CONTAINS DECIMAL POINT.

A numeric string that appears to be a numeric procedure name contains a decimal point. The compiler ignores the presence of the decimal point and proceeds with the processing of the numeric procedure name.

1076 .RELATIVE KEY. ITEM DEFINED IN RECORD OF FILE.

A data-name referenced in a RELATIVE KEY phrase of a SELECT clause is defined in the record description of the associated file. The compiler ignores this error and continues to process the RELATIVE KEY data-name.

1077 NO. OF AREAS DEFAULTS TO MAX. FOR FILE TYPE.

The number of buffer areas reserved by the RESERVE clause is greater than the maximum allowed for the file organization. The compiler allocates two areas for a sequential file and one for a relative file.

1105 UNRECOGNIZED LITERAL TYPE...SYSTEM ERROR

The compiler has failed to properly identify a literal. System error. Fatal.

1107 .TO. OR .GIVING. MISSING IN ADD

The keyword TO or GIVING was not found after the second operand in an ADD statement. Fatal.

1110 MORE THAN 18. DIGITS IN COMPOSITE. TRUNCATING.

The length of an arithmetic composite is greater than 18 digits. The composite is truncated on the left to 18 digits. Warning.

1111 ONLY ONE DEST ALLOWED AFTER .CORRESPONDING. USE FIRST.

More than one destination data-name follows the keyword CORRESPONDING. The compiler ignores all but the first. Warning.

1113 UNSIGNED COMP 3 ITEMS ILLEGAL

The PICTURE for a COMP-3 item does not contain an S character. Fatal.

1114 ARGUMENT CANNOT BE PASSED .BY DESCRIPTOR.

The compiler detected an identifier with COMPUTATIONAL usage, SEPARATE SIGN, or JUSTIFIED RIGHT that is being passed by DESCRIPTOR in the USING phrase of a CALL statement. Such items can be passed only by REFERENCE or VALUE. Fatal.

1115 .BY VALUE. ARG. MUST BE .COMP. LONGWORD INTEGER.

An argument passed by VALUE in a CALL statement must be a data item that is declared with COMPUTATIONAL usage with no V character in its PICTURE string. The item must have from five to nine decimal positions; that is, its PICTURE must be in the range 9(5) to 9(9). Fatal.

1116 ARGUMENT OMITTED AFTER .BY. CLAUSE IN .CALL.

The compiler detected no identifier following the keywords BY REFERENCE, BY VALUE, or BY DESCRIPTOR in a CALL statement. Fatal.

1117 .GIVING. ITEM MUST BE .COMP. LONGWORD INTEGER.

The identifier following the keyword GIVING in a CALL statement must be declared with COMPUTATIONAL usage with no V character in its PICTURE string. The item must have from five to nine decimal positions; that is, its PICTURE must be in the range 9(5) to 9(9). Fatal.

1121 .SEARCH. VERB NOT PROCESSED

Because of an earlier diagnostic (warning or fatal), the compiler cannot process the SEARCH statement completely. Fatal.

APPENDIX E

RUN-TIME ERROR MESSAGES

This appendix describes error messages that are produced by the COBOL-74 run-time system (RTS). Run-time messages look like this:

%C74-F-CODE, TEXT [string-1] [(string-2)] [string-3]

where:

%C74	is the facility code for VAX-11 COBOL-74
F	is the severity level (fatal). All RTS errors are fatal.
CODE	is the mnemonic composed of the first three characters of the first three words of the RTS error message.
TEXT	is the body of the error message
string-1	is the character string specified in VALUE OF ID clause
(string-2)	is the literal used in SELECT...ASSIGN entry
string-3	is the RMS error message, this messages is printed on the line after the error message itself.

Associated with each error is an eight digit hexadecimal status code (enclosed in parenthesis). The status code is not printed with the run-time message, however. You can use this code to check the completion status of DCL commands within an indirect command procedure or batch file. For example, consider the following program:

```
00001    IDENTIFICATION DIVISION.
00002    PROGRAM-ID.
00003            SUBERR.
00004    ENVIRONMENT DIVISION.
00005    SOURCE-COMPUTER. VAX-11.
00006    OBJECT-COMPUTER. VAX-11.
00007    SPECIAL-NAMES.
00008            CONSOLE IS CONSOL.
00009    DATA DIVISION.
00010    WORKING-STORAGE SECTION.
00011    01 T-NUMBER                          PIC 9.
00012    01 TABLE-VALUES.
00013            03  FILLER                  PIC X(5) VALUE "ONE  ".
00014            03  FILLER                  PIC X(5) VALUE "TWO  ".
00015            03  FILLER                  PIC X(5) VALUE "THREE".
00016            03  FILLER                  PIC X(5) VALUE "FOUR ".
00017            03  FILLER                  PIC X(5) VALUE "FIVE ".
00018    01 T-VALUES REDEFINES TABLE-VALUES.
00019            03  TAB-VAL OCCURS 5 TIMES INDEXED BY T-INDEX.
00020                05  FILLER              PIC XXXXX.
00021    PROCEDURE DIVISION.
00022    MAIN SECTION.
00023    PARA.
00024            DISPLAY "ENTER NUMBER : " UPON CONSOL
00025                    NO ADVANCING.
00026            ACCEPT T-NUMBER FROM CONSOL.
00027            IF T-NUMBER = 0
00028                    GO TO FINISH.
00029            SET T-INDEX TO T-NUMBER.
00030            DISPLAY TAB-VAL(T-INDEX) UPON CONSOL.
00031            GO TO PARA.
00032    FINISH.
00033            DISPLAY "*** END OF SESSION ***" UPON CONSOL.
00034            STOP RUN.
```

The program will prompt you for input - numbers from 1 to 5 (0 will cause the program to terminate). If you enter any other number, the program will abort, and the RTS will print "Subscript or index out of range". If you run SUBERR from your terminal, you could restart the program and continue. If you run SUBERR as part of a command procedure or batch job you could not restart the program from your terminal. You can, however, include commands in the file to test the completion status of the program and take action:

```
$ ASSIGN 'P1 COB$CONSOLE
$ SET NOON
$ AGAIN:
$ RUN SUBERR
$ IF $STATUS .EQ. %X101DB00C THEN GOTO WRITER
$ EXIT
$ WRITER:
$ WRITE SYS$OUTPUT "Index must be less than 6 - type 0 to quit"
$ GOTO AGAIN
```

NOTE

The parameter 'P1 in command line "ASSIGN..." allows you to specify a device at run-time.

If you include the preceding commands in a file named CHECKER.COM, link SUBERR with the /NOTRACEBACK qualifier, and execute the procedure, the following will appear at your terminal:

```
@CHECKER OPA0:
ENTER NUMBER :
```

You could now input a number from the system console:

```
ENTER NUMBER : 3
THREE
ENTER NUMBER :
```

If you enter a number other than 0 through 5, the following will appear on the terminal:

```
ENTER NUMBER : 3
THREE
ENTER NUMBER : 6
%C74-F-SUBOUTRAN, subscript out of range
Index must be less than 6 - type 0 to quit
ENTER NUMBER :
```

See the **VAX/VMS Command Language User's Guide** for a full discussion of command procedures and batch jobs.

The remainder of this appendix describes the run-time error messages.

BADFILNAM, bad file name string-1 (string-2)
(001DB0E4)

The file specification and/or associated switches for a file description are syntactically incorrect.

CLOERRFIL, CLOSE error on file string-1 (string-2) string-3
(001D8094)

The execution of a CLOSE statement failed. The accompanying RMS error code further specifies the error.

COMGENABO, compiler generated abort
(001D8034)

The run-time system tried to execute a part of the program that contains fatal errors. See your source program listing.

DELERRFIL, DELETE error on file string-1 (string-2) string-3
(001D80CC)

The execution of a DELETE statement failed. The accompanying RMS error code further specifies the error.

ERRACC, error in ACCEPT logical-name string-3
(001D810C)

An error occurred while trying to ACCEPT a record from the file associated with logical-name. The RMS error code further specifies the error.

"SHOW TRANSLATION logical-name" can be use to determine the value of logical-name.

ERRCLOUNI, error in CLOSE UNIT or REEL on file
(001D80B4) string-1 (string-2) string-3

The program executed a CLOSE UNIT or CLOSE REEL statement that failed. The accompanying RMS error code further specifies the error.

ERRDIS, error in DISPLAY logical-name string-3
(001D8104)

An error occurred while trying to DISPLAY a record from the file associated with logical-name. The RMS error code further specifies the error.

EXPTOOLAR, exponent too large
(001D8054)

The exponent used in a COMPUTE statement is out of range. The legal range is -32768 to 32767.

FILALROPN, file string-1 already open
(001D805C)

The program tried to open a file that is currently open.

FILPRELOC, file string-1 (string-2) previously locked
(001D807C)

The program tried to access a file for which it had previously executed a CLOSE...WITH LOCK statement.

FINNOTOPN, file string-1 not open
(001D8064)

The program tried to CLOSE or otherwise access a file that is not currently open.

INVDECDAT, invalid data in decimal data-item <address-of-error>
(001D8114)

Numeric DISPLAY item contains invalid data value, such as non-numeric character or invalid separate signs. DISPLAY items must be initialized before use.

The INSPECT statement can be used to clean-up invalid data in existing files.

INVLINVAL, invalid LINAGE value on file string-1 (string-2)
(001D80A4)

The LINAGE clause specifies a page body size that results in an invalid value; the value is not greater than zero, or it is out of range.

INVOPEFIL, invalid operation on file string-1 (string-2)
(001D806C)

The program tried to execute one of the following I/O statements for a file that is open in an incompatible mode:

(a) a READ for a file open for OUTPUT

(b) a WRITE for a file open for INPUT

(c) an I/O operation not consistent with the file organization (for example, START on a sequential file).

NOEOFPRO, no EOF processing on file string-1 (string-2)
(001D80AC)

An end-of-file condition has been detected, but the I/O statement does not have an AT END clause, and the program has no USE procedure for end-of-file processing.

NOTPONOPR, file string-1 (string-2) not open for operation
(001D809C)

The program tried to execute an I/O statement for a file that is not open.

NULGOTO, GO TO executed has not been ALTERed
(001D802C)

The program reached an alterable GO TO statement before assigning it a procedure name.

NUMARGCAL, number of arguments in CALL incorrect
(001D8044)

The number of arguments received by a COBOL subprogram does not agree with the expected number of arguments; that is, the number of CALL statement arguments in the calling program is not the same as the number of arguments in the PROCEDURE DIVISION USING phrase of the called program.

OCCDEPVAL, OCCURS DEPENDING value out of range
(001D8014)

The value of the data item that defines the size of the table is not in the table size range specified in the OCCURS clause.

OPNCOBIN, error opening logical-name string-3
(001D80FC)

An error occurred while trying to open for ACCEPT the file associated with logical-name. The RMS error code further specifies the error.

OPNCOBOUT, error opening logical-name string-3
(001D80F4)

An error occurred while trying to open as DISPLAY the file associated with logical-name. The RMS error code further specifies the error.

OPNERRFIL, OPEN error on file string-1 (string-2) string-3
(001D808C)

The execution of an OPEN statement failed. The accompanying RMS error code further specifies the error.

PERCOUTOO, PERFORM counter too large
(001D804C)

The value of the iteration counter used in a PERFORM statement exceeded 32767.

REAERRFIL, READ error on file string-1 (string-2) string-3
(001D80B4)

The execution of a READ statement failed. The accompanying RMS error code further specifies the error.

REANOTPRE, read not preceded by rewrite or delete on file
(001D8074) string-1 (string-2)

The program attempted to execute a REWRITE or DELETE statement for a sequentially accessed file, but the last I/O operation on the file was not a READ.

RECCALDET, recursive CALL detected
(001D803C)

A COBOL subprogram attempted to call itself, either directly or indirectly. The EXIT PROGRAM statement must be executed in a subprogram before the subprogram can be called again.

RECPERDET, recursive PERFORM detected
(001D8024)

The program attempted to execute a PERFORM statement whose exit is also the exit of a previously executed PERFORM that is still active.

RETEXTOUT, PERFORM exited out of order
(001D801C)

The program reached the end of an active PERFORM while processing a more recently executed PERFORM; that is, the program executed a PERFORM statement whose range overlaps the end of the PERFORM statement that is currently being executed.

REWERRFIL, REWRITE error on file string-1 (string-2) string-3
(001D80C4)

The execution of a REWRITE statement failed. The accompanying RMS error code further specifies the error.

SAMAREUSE, SAME AREA already in use when opening file string-1 (string-2)
(001D80EC)

The program tried to OPEN a file that uses the same buffer area as another open file.

STAERRFIL, START error on file string-1 (string-2) string-3
(001D80D4)

The execution of a START statement failed. The accompanying RMS error code further specifies the error.

SUBOUTRAN, subscript or index out of range
(001D800C)

The subscript value for a data item, or the value of an index-name, is not greater than zero, or it is greater than the maximum number of occurrences of the table data item.

UNLERRFIL, UNLOCK error on file string-1 (string-2) string-3
(001D80DC)

An unsuccessful attempt has been made to unlock a record in the file. The accompanying RMS error code further specifies the error.

WRIERRFIL, WRITE error on file string-1 (string-2) string-3
(001D806C)

The execution of a WRITE statement failed. The accompanying RMS error code further specifies the error.

APPENDIX F

INTERNAL COMPILER ERRORS -- SYSTEM ERRORS

This appendix lists errors that the VAX-11 COBOL-74 compiler displays if it aborts.

System errors have the following form:

C74 -- SYSTEM ERRORXXXXXX

where:

XXXXXX is a six digit error code.

ERROR MESSAGE

000000 NO MORE AVAILABLE DISK SPACE

The Work File system has exhausted all available disk space.

a. The compiler uses the longest contiguous disk area available up to 1024 64-word blocks. If less than 1000 blocks remain, clear the disk and start again.

b. If 1000 blocks are available, submit a smaller job.

000604 UNRECOGNIZABLE SOURCE FILE

The compiler could not process the source file because:

a. /ANSI_FORMAT qualifier was not used, and source program is not in terminal reference format.

b. IDENTIFICATION missing

The following errors represent internal table overflow in the compiler:

000017 TOO MANY PROCEDURE NAMES

000020 TOO MANY PROCEDURE NAMES

000065 TOO MANY LITERALS

000077 COMMAND BLOCK TOO LARGE

a. A Procedure Division statement is too large for the compiler to handle.

b. Reduce the number of identifiers used in the statement.

000115 DATA NAME TABLE OVERFLOW

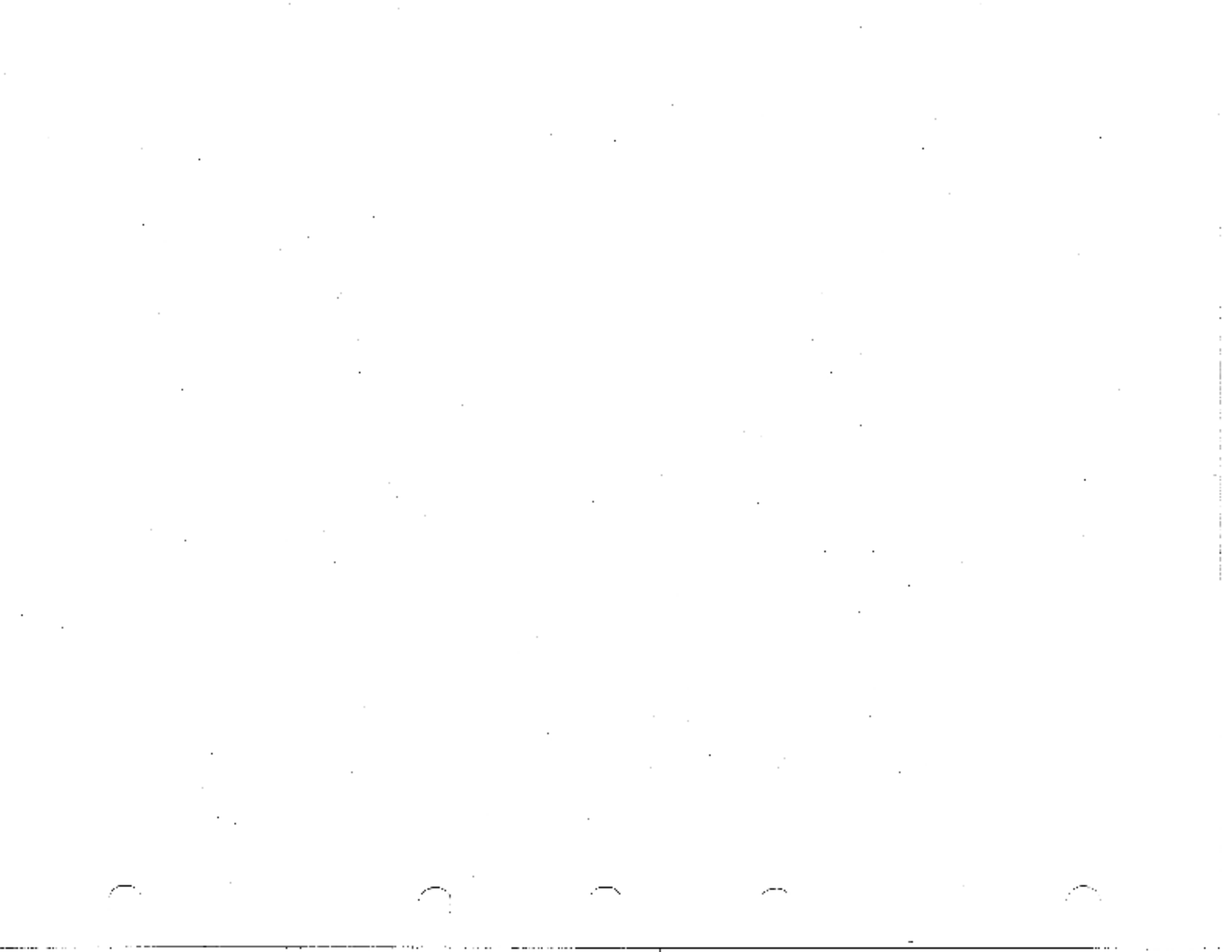
The following errors occur if the operating system or the COBOL-74 compiler malfunctions. You should submit a Software Performance Report (SPR) for any of these errors.

NOTE

You should include the source program in machine readable form for any SPR you submit.

000001 INTERNAL WORK FILE ERROR
000002 INTERNAL WORK FILE ERROR
000003 INTERNAL WORK FILE ERROR
000004 BAD USAGE CODE
000005 ELT ALREADY DEALLOCATED
000011 INSUFFICIENT CORE
000012 INSUFFICIENT CORE
000013 INTERNAL ALLOCATION ERROR
000014 INTERNAL WORK FILE ERROR
000015 ATTEMPT TO PROCESS MORE ELSE'S THAN IF'S
000016 INVALID FILE CLASS
000021 ERROR IN PROCESSING PROCEDURE TAG TABLE
000023 ERROR IN PROCESSING PROCEDURE TAG TABLE
000033 WORK FILE SEARCH ERROR
000036 INDEX NAME UNDEFINED BY COMPILER
000037 INTERNAL ERROR
000053 WORK FILE SEARCH ERROR
000063 DATA DIVISION SEQUENCE ERROR
000070 ERROR IN SUBSCRIPT PROCESSING
000072 DATA ITEM STARTING ON ODD ADDRESS
000075 ERROR IN DISPLAY DATA ITEM
000076 EXCEEDED STACK CAPACITY

000100	COMPILER LOOP: TOO MANY SUBORDINATE ITEMS
000101	RECORD PROCESSING LOGIC ERROR
000102	ERROR IN ITEM PROCESSING LOGIC
000104	INVALID USAGE CODE
000105	ILLEGAL LEVEL NUMBER
000106	LEVEL 01 READ BUT DOES NOT VALIDATE
000111	ERROR IN EDITED ITEM
000112	ILLEGAL TYPE DATA ITEM
000116	RESERVED WORD TABLE ERROR
000120	DELIMITED SENDING ITEM NOT FOUND FOR STRING CMD
000121	ERROR IN DELIMITED SENDING ITEM FOR STRING CMD
000122	SENDING ITEM NOT FOUND FOR STRING CMD
000123	ERROR IN TALLYING DATA ITEM FOR UNSTRING
000124	ERROR IN DELIMITED RECEIVING ITEM FOR UNSTRING
000125	ERROR IN RECEIVING ITEM FOR UNSTRING
000126	ERROR IN DELIMINTED SENDING ITEM FOR UNSTRING
000127	DELIMITED SENDING ITEM FOR UNSTRING
000130	DELIMITED SENDING ITEM FOR UNSTRING
000131	INSPECT TALLYING ERROR
000132	INSPECT TALLYING BEFORE/AFTER ERROR
000133	INSPECT REPLACING ERROR
000134	INSPECT REPLACING ERROR
000135	INSPECT REPLACING BEFORE/AFTER INITIAL ERROR
000140	EXTENDED NAME TABLE ERROR
000140	SAME AREA RECORD CLAUSE ERROR
000141	SAME AREA RECORD CLAUSE ERROR
000150	INTERNAL STACK ERROR
000151	CORRESPONDING ERROR
000201	STACK ERROR MESSAGE
000201	PROCESSING ERROR OCCURS DEPENDING ON CLAUSE
000202	PRT KEY ERROR
000204	INDEX DATA ITEM NOT FOUND
000220	ERROR IN DESTINATION FOR SEARCH
000221	WORK FILE READ ERROR
000230	INVALID IBF FLAG READ
000231	WORK FILE READ SEQ. ERROR
000232	ERROR IN ALTER PROCESSING
000233	INVALID LINKAGE SECTION ITEM
000234	WORK FILE SEARCH ERROR
000997	INTERNAL ERROR
000999	INTERNAL ERROR
005000	ILLEGAL OPERATOR FOR BOOLEAN OPERATION
005001	ILLEGAL USAGE FOR OPERANDS IN RELATIONAL CODE
005002	ILLEGAL USAGE FOR OPERANDS IN RELATIONAL CODE
005003	ILLEGAL NUMERIC COMPARISON
005004	ILLEGAL LITERAL IN ARITHMETIC OPERATION
005005	ILLEGAL USAGE FOR DIVIDE OPERAND
005006	DIVIDE ERROR
005007	ILLEGAL USAGE FOR MULTIPLY OPERAND
005010	MULTIPLY ERROR
005011	ILLEGAL USAGE FOR ARITHMETIC OPERAND
005012	ADD/SUBTRACT ERROR
005013	ILLEGAL ARITHMETIC OPERATOR
005014	ILLEGAL COMPARE FOR SEARCH ALL
005015	INDEX PROBLEM IN TABLE



APPENDIX G

PROGRAMMING EXAMPLES

This appendix contains examples of VAX-11 COBOL-74 programs. The programs do not represent real applications; they are intended to suggest useful techniques and their implementation with COBOL-74.

G.1 CALLING A FORTRAN SUBROUTINE

COBOL-74 modules can be linked with other VAX-11 native-mode modules, even if they were written in other languages. This capability is often useful, especially when a feature is not available in COBOL-74, but is available in another language. One example is the square root function, which is not available in COBOL-74 (non-integer exponents are not allowed).

G.1.1 The COBOL Program, GETROOT

This program accepts a value from the terminal, calls the FORTRAN subroutine, SQROOT, and passes the value as a character string (BY DESCRIPTOR, because that is how FORTRAN expects it). It then displays the result.

COBOL-74 does not support the floating point data type; therefore, the FORTRAN subroutine returns the result as a character string. Note that because the result is a character string with a decimal point, the program uses a MOVE CORRESPONDING to place it in a numeric data item. The second DISPLAY shows the result of editing the returned value.

An INSPECT statement replaces all space characters in the result by zeros; space characters would cause a reserved operand fault when the MOVE statement was executed.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
    GETROOT.
INSTALLATION.    DIGITAL EQUIPMENT CORPORATION.

*********************************************
*                                           *
* This program demonstrates the use of the  *
* CALL statement in calling FORTRAN-IV-PLUS  *
* programs. It calls a FORTRAN subprogram    *
* which returns the square root of the       *
* argument.                                 *
*                                           *
*********************************************

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.    VAX-11.
OBJECT-COMPUTER.    VAX-11.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  INPUT-NUMBER.
    03  INTEGER        PIC 9(5).
    03  DEC-POINT      PIC X(1).
    03  DECIMAL        PIC 9(8).

01  WORK-NUMBER.
    03  INTEGER        PIC 9(5).
    03  DECIMAL        PIC 9(8).
01  WORK-NUMBER-A REDEFINES WORK-NUMBER  PIC 9(5)V9(8).

01  DISPLAY-NUMBER     PIC ZZ,ZZ9.9999.

PROCEDURE DIVISION.
STARTER SECTION.
SBEGIN.
    MOVE SPACES TO INPUT-NUMBER.
    DISPLAY "Enter number (with decimal point): "
      NO ADVANCING.
    ACCEPT INPUT-NUMBER.
    IF INPUT-NUMBER = SPACES
      GO TO ENDJOB.
    CALL "SQROOT" USING BY DESCRIPTOR INPUT-NUMBER.
    IF INPUT-NUMBER = ALL "*"
      DISPLAY "** INVALID ARGUMENT FOR SQUARE ROOT"
    ELSE
      DISPLAY "The square root is: " INPUT-NUMBER
      INSPECT INPUT-NUMBER
        REPLACING ALL " " BY "0"
      MOVE CORRESPONDING INPUT-NUMBER TO WORK-NUMBER
      MOVE WORK-NUMBER-A TO DISPLAY-NUMBER
      DISPLAY DISPLAY-NUMBER.
    GO TO SBEGIN.
ENDJOB.
    STOP RUN.
```

G.1.2 The FORTRAN Program, SQROOT

This subroutine accepts a 14-character string, decodes it into a real variable (DECODE is analogous to an internal READ). It then calls the SQRT function in the statement that encodes the result into the 14-character argument.

```
      SUBROUTINE SQROOT(ARG)
      CHARACTER*14 ARG
      DECODE(14,10,ARG,ERR=20)VAL
10    FORMAT(F12.6)
      IF(VAL.LE.0.)GO TO 20
      ENCODE(14,10,ARG)SQRT(VAL)
999   RETURN
20    ARG='**************'
      GO TO 999
      END
```

G.1.3 Sample Run of GETROOT

```
$ RUN GETROOT <CR>
Enter number (with decimal point): 25. <CR>
The square root is:      5.000000
     5.0000
Enter number (with decimal point): HELLO <CR>
** INVALID ARGUMENT FOR SQUARE ROOT
Enter number (with decimal point): 1000000. <CR>
The square root is:  1000.000000
 1,000.0000
Enter number (with decimal point): 2. <CR>
The square root is:      1.414214
     1.4142
Enter number (with decimal point): <CR>
$
```

G.2 CALLING VAX-11 RUN-TIME PROCEDURES

The VAX-11 Common Run-time Procedure Library contains sets of general purpose and language-specific procedures. The procedures are written using the VAX-11 procedure calling standard; therefore, they are callable by COBOL-74.

The procedures are described in the VAX-11 Common Run-Time Procedure Library Reference Manual.

G.2.1 The COBOL Program, RUNTIME

This program calls two procedures in the Run-Time Library: LIB$MOVTC and FOR$DATE.

LIB$MOVTC uses a translation table to translate each character in a character string from one code form to another (EBCDIC to ASCII, for example). This program uses LIB$MOVTC to translate all lower-case characters to upper case and all non-graphic characters to spaces. Note that LIB$MOVTC expects all parameters to be passed by descriptor.

FOR$DATE returns the system date as a 9-character string. Note that it expects the string to be passed by reference.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  RUNTIME.

*************************************************
*                                               *
*  This program demonstrates the method for     *
*  calling VAX-11 Run-time Procedures.          *
*                                               *
*************************************************

DATE-COMPILED.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.    VAX-11.
OBJECT-COMPUTER.    VAX-11.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  INPUT-AREA PIC X(50).
01  OUTPUT-AREA PIC X(50).
01  FILL-CHAR PIC X VALUE SPACE.
01  TRANSLATION-TABLE.
    03  FILLER PIC X(32) VALUE SPACES.
    03  FILLER PIC X(24) VALUE " !""#$%&'()*+,-./01234567".
    03  FILLER PIC X(24) VALUE "89:;<=>?@ABCDEFGHIJKLMNO".
    03  FILLER PIC X(24) VALUE "PQRSTUVWXYZ[\]^_`ABCDEFG".
    03  FILLER PIC X(24) VALUE "HIJKLMNOPQRSTUVWXYZ{|}~ ".

01  DATE-AREA PIC X(9).
```

```
PROCEDURE DIVISION.
STARTIT SECTION.
BEGIN.

    DISPLAY "Enter string:".
    MOVE SPACES TO INPUT-AREA.
    ACCEPT INPUT-AREA.

    CALL "LIB$MOVTC" USING BY DESCRIPTOR
        INPUT-AREA
        FILL-CHAR
        TRANSLATION-TABLE
        OUTPUT-AREA.

    DISPLAY OUTPUT-AREA.

    CALL "FOR$DATE" USING DATE-AREA.
    DISPLAY DATE-AREA.

    STOP RUN.
```

G.2.2 Sample Run of RUNTIME

```
$ RUN RUNTIME
Enter string:
How do I love thee? Let me count... 8, 9, A
HOW DO I LOVE THEE? LET ME COUNT... 8, 9, A
18-DEC-78

$
```

G.3 USING TERMINAL ESCAPE SEQUENCES

It is often useful to design terminal screen forms for data input applications. You can implement forms on most terminals by using escape sequences.

G.3.1 The COBOL Program, ESCAPE

This program accepts data from a DIGITAL VT52 terminal; it builds a form and guides the operator by using escape-character sequences. Other terminals have similar capabilities.

The program defines the ESC character (decimal 27, hex 1B) as a one-word COMPUTATIONAL item, and redefines it as a one-byte data item. The same technique is used for the one-character row and column values required for direct cursor addressing. (Chapter 4 describes the internal format of COMP data items.)

The program contains a table that describes the format of the terminal screen. First, the program clears the screen. It then cycles through the table to display the prompts. During a second pass through the table, the program "paints" the input area for each field with underscore characters, and accepts data from the operator.

Note the use of the OCCURS ... DEPENDING ON clause in the description of INPUT-AREA, allowing the variable-length display.

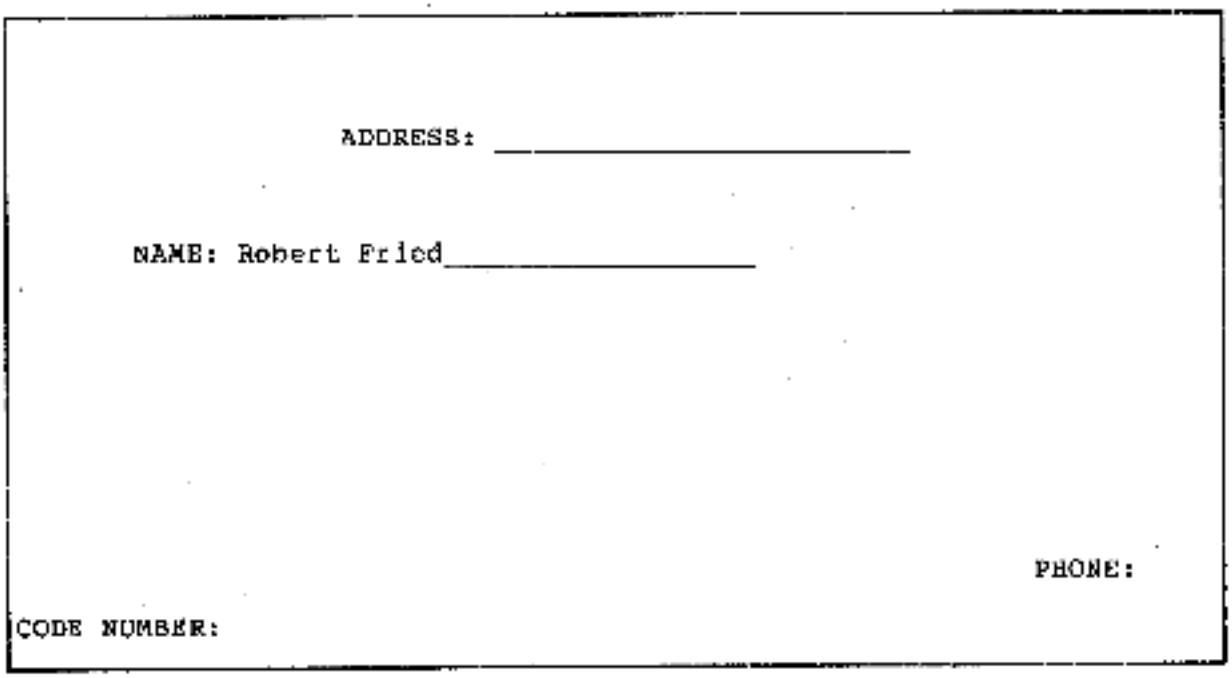
```
IDENTIFICATION DIVISION.
PROGRAM-ID.
    ESCAPE.

*************************************************
*                                               *
*  This program demonstrates the use of terminal *
*  escape sequences, including cursor control,   *
*  through COBOL.                               *
*                                               *
*************************************************

DATE-COMPILED.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.    VAX-11.
OBJECT-COMPUTER.    VAX-11.
DATA DIVISION.
WORKING-STORAGE SECTION.
01    INPUT-AREA.
    03  IACHAR PIC X(1) OCCURS 1 TO 30 TIMES
        DEPENDING ON IASUB.
01    IASUB PIC 9(2) COMP.

01    FIELD-TABLE.
    03  FILLER PIC X(24) VALUE "050830NAME:".
    03  FILLER PIC X(24) VALUE "012024ADDRESS:".
    03  FILLER PIC X(24) VALUE "180106CODE NUMBER:".
    03  FILLER PIC X(24) VALUE "166010PHONE:".
01    FIELD-TAB-A REDEFINES FIELD-TABLE.
    03  FIELD-TAB-ENTRY  OCCURS 4 TIMES.
        05  ROW          PIC 9(2).
        05  COLUM        PIC 9(2).
        05  FIELD-LENGTH    PIC 9(2).
        05  PROMPT       PIC X(18).
```

```
01     ROW-VALUE PIC 9(3) COMP.
01     ROW-VALUE-C REDEFINES ROW-VALUE PIC X(1).

01     COLUM-VALUE PIC 9(3) COMP.
01     COLUM-VALUE-C REDEFINES COLUM-VALUE PIC X(1).

01     ESCAPER PIC 9(3) COMP VALUE 27.
01     ESCAPE-C REDEFINES ESCAPER PIC X(1).

01     CTR PIC 9(2).

PROCEDURE DIVISION.
LOOP.

* ESC H moves the cursor to the home position.
* ESC J clears from the cursor position to the
*       end of the screen.

    DISPLAY ESCAPE-C "H" ESCAPE-C "J".
    PERFORM PROC-FIELD VARYING CTR FROM 1 BY 1
      UNTIL CTR > 4.
    PERFORM ACCEPT-FIELD VARYING CTR FROM 1 BY 1
      UNTIL CTR > 4.
    IF CTR < 90
      GO TO LOOP.
    STOP RUN.

PROC-FIELD.

* Compute the row and column values from the table.
* Also, set the OCCURS DEPENDING item to the length
*   of the prompt.

    PERFORM COMP-LEN.

* Display the prompt.

    DISPLAY ESCAPE-C "Y"
      ROW-VALUE-C COLUM-VALUE-C
      INPUT-AREA NO ADVANCING.

ACCEPT-FIELD.

* Recompute the prompt location.

    PERFORM COMP-LEN.

* Position the cursor at the end of the prompt.

    ADD IASUB TO COLUM-VALUE.

* Set the OCCURS DEPENDING item to the length of
*   the expected input entry.

    MOVE FIELD-LENGTH(CTR) TO IASUB.
```

```
* Display a string of underscores the same length
*   as the expected input entry.

    MOVE ALL "_" TO INPUT-AREA.
    DISPLAY ESCAPE-C "Y"
      ROW-VALUE-C COLUM-VALUE-C
      " " INPUT-AREA.

* Reposition the cursor to the beginning of the
*   input area.

    DISPLAY ESCAPE-C "Y"
      ROW-VALUE-C COLUM-VALUE-C
      " " NO ADVANCING.

* Set the OCCURS DEPENDING item to make INPUT-AREA
*   its maximum size.

    MOVE 30 TO IASUB.
    MOVE SPACES TO INPUT-AREA.

* Get the input.

    ACCEPT INPUT-AREA.

* Input processing code goes here.

    IF INPUT-AREA = "QUIT"
      MOVE 91 TO CTR.

DONOTHING.
    EXIT.

COMP-LEN.

* Set the OCCURS DEPENDING item to make INPUT-AREA
*   its maximum size.

    MOVE 30 TO IASUB.

* Escape sequence row and column values begin with 32,
*   so we add 31 to the row and column numbers.

    ADD 31 ROW(CTR) GIVING ROW-VALUE.
    ADD 31 COLUM(CTR) GIVING COLUM-VALUE.

* Move the prompt to the display area, then determine
*   its length by locating the last character (:).

    MOVE PROMPT(CTR) TO INPUT-AREA.
    PERFORM DONOTHING VARYING IASUB FROM 1 BY 1
      UNTIL IACHAR(IASUB) = ":".
```

G.3.2 Sample Run of ESCAPE

In this example of a screen during the execution of ESCAPE, the program has just accepted the name and is waiting for the operator to enter the address.



G.4 CALLING VAX/VMS SYSTEM SERVICES

System services are procedures that the VAX/VMS operating system uses to control resources available to processes, to allow communication among processes, and to perform basic operating system functions.

Although most system services are used primarily by the operating system, you can use many of them yourself in VAX-11 COBOL-74 programs. System services are described in the **VAX/VMS System Services Reference Manual**.

G.4.1 The COBOL Program, SYSTSVC

This program calls two system services: $GETMSG and $TRNLOG.

$GETMSG returns the text of a system message to the caller, using the unique identification number that is assigned to each system message. This program uses $GETMSG to translate the return status from the other system service, $TRNLOG.

$TRNLOG translates a logical name and returns the equivalence name string. It also returns one of five return status codes.

The program accepts a logical name from the terminal. It then calls $TRNLOG (note that the prefix, SYS, is required) to get the equivalence name. Because the equivalence name could itself be a logical name, the program checks the return status; it repeatedly calls $TRNLOG until the return status indicates that no translation occurred.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
    SYSTSVC.
INSTALLATION.    DIGITAL EQUIPMENT CORPORATION.

*****************************************
*                                       *
* This program demonstrates the use of the *
* CALL statement in calling VAX/VMS System *
* Services.                             *
*                                       *
*****************************************

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.    VAX-11.
OBJECT-COMPUTER.    VAX-11.
```

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  LOG-STRING.
    03  LOG-CHAR PIC X OCCURS 1 TO 30 TIMES
         DEPENDING ON LOG-LENGTH.
01  LOG-LENGTH PIC 9(2).
01  LOG-NAME PIC X(30).
01  IN-PTR PIC 9(2).
01  NAME-LENGTH PIC S9(4) COMP.
01  EQUIV-NAME PIC X(30) VALUE SPACES.
01  PHYSICAL-TEST REDEFINES EQUIV-NAME.
    03  PT-1-4.
        05  PT-CHAR  PIC X.
        05  FILLER PIC X(3).
    03  PHYSICAL-NAME PIC X(26).
01  ESCAPE-CHAR PIC 9(3) COMP VALUE 176.
01  ESCAPE REDEFINES ESCAPE-CHAR PIC X.
01  RES PIC S9(8) COMP.
    88  SUCCESSFUL  VALUE 1.
01  RESULT-NAME  PIC X(30).
01  OUT-LENGTH PIC Z(4).
01  RESULT-DISP PIC 9(8).
01  DUMMY-ARG PIC S9(9) COMP VALUE 0.

01  MESSAGE-AREA.
    03  MESSAGE-CHAR OCCURS 1 TO 256 TIMES
          DEPENDING ON MESSAGE-LENGTH  PIC X.
01  MESS-LEN  PIC 9(8) COMP.
01  MESSAGE-LENGTH PIC 9(8) COMP.
01  MASK PIC 9(8) COMP VALUE IS 15.
01  MSG-VALUE PIC X(4).

PROCEDURE DIVISION.
STARTER SECTION.
SBEGIN.
    DISPLAY "ENTER LOGICAL NAME: " NO ADVANCING.
    ACCEPT LOG-NAME.
SLOOP.
    PERFORM TRANSLATE-LOGICAL-NAME.
    PERFORM GET-RESULT.
    IF PT-CHAR = ESCAPE
      MOVE PHYSICAL-NAME TO RESULT-NAME
    ELSE
      MOVE EQUIV-NAME TO RESULT-NAME.
    MOVE NAME-LENGTH TO OUT-LENGTH.
      DISPLAY LOG-NAME "= " RESULT-NAME.

*          If the translation was performed,
*          use the result to attempt further
*          translation.

    IF  SUCCESSFUL
      MOVE RESULT-NAME TO LOG-NAME
      GO TO SLOOP.
    STOP RUN.
```

```
*************************************************
*                                               *
* The following paragraph uses the contents of  *
* LOG-STRING as an argument to the SYS$TRNLOG   *
* System Service. The translated name is placed *
* in EQUIV-NAME and the length of the result    *
* is in RES-LEN. RES contains the status code   *
* from SYS$TRNLOG.                              *
*                                               *
*************************************************

TRANSLATE-LOGICAL-NAME.
    MOVE SPACES TO EQUIV-NAME.

*           Initialize UNSTRING pointer

    MOVE 1 TO IN-PTR.

*           Initialize DEPENDING ON subscript

    MOVE 30 TO LOG-LENGTH.

*           Create string of exact length

    UNSTRING LOG-NAME DELIMITED BY " "
      INTO LOG-STRING WITH POINTER IN-PTR.

*           Fix up string length from pointer.

    SUBTRACT 2 FROM IN-PTR GIVING LOG-LENGTH.

*           Delete colon from logical-name string.

    IF LOG-CHAR (LOG-LENGTH) = ":"
      SUBTRACT 1 FROM LOG-LENGTH.
    CALL "SYS$TRNLOG"
      USING BY DESCRIPTOR LOG-STRING,
        BY REFERENCE NAME-LENGTH
        BY DESCRIPTOR EQUIV-NAME
        BY VALUE DUMMY-ARG, DUMMY-ARG, DUMMY-ARG
          GIVING RES.

GET-RESULT SECTION.
SBEGIN.
    MOVE 256 TO MESSAGE-LENGTH.
    MOVE SPACES TO MESSAGE-AREA.
    CALL "SYS$GETMSG"
      USING
        BY VALUE RES
        BY REFERENCE MESS-LEN
        BY DESCRIPTOR MESSAGE-AREA
        BY VALUE MASK
        BY DESCRIPTOR MSG-VALUE.
    MOVE MESS-LEN TO MESSAGE-LENGTH.
    DISPLAY MESSAGE-AREA.
```

G.4.2 Sample Run, SYSTSVC

```
$ RUN SYSTSVC
ENTER LOGICAL NAME: COB$INPUT
%SYSTEM-S-NORMAL, normal successful completion
COB$INPUT                       = SYS$INPUT
%SYSTEM-S-NORMAL, normal successful completion
SYS$INPUT                       = TTE1:
%SYSTEM-S-NOTRAN, no string translation performed
TTE1:                         = TTE1

$
```

ACCEPT, 6-26, 6-27, 6-28
Access mode, 6-14
ADD, 4-17
  multiple operands, 4-16
/AL:n, file specification
  switch, 6-22
Alphabetic class, 3-1
Alphanumeric class, 3-1
ALTER, 7-5
Ambiguities, DEBUG, 9-5
/ANSI_FORMAT, 2-3
APPLY, 6-17
Arithmetic expression
  processing, 4-21
  temporary work area, 4-21
Arithmetic statements, 4-13
ASSIGN, 6-20
ASSIGN command, 2-12, 6-19, 6-27
AT END, 6-16, 6-26, 10-7

Binary, 4-1
Block, 6-5
  physical, 6-5
Blocking, 6-5, 13-9
Breakpoint, DEBUG, 9-5, 9-6
/BRIEF, LINK qualifier, 2-9
Bucket, 6-5, 13-6
Buffer defaults, RMS, 6-13
Buffers, 6-13
  multiple, 6-13
  sharing, 6-14

Caching index roots, 13-8
CALL, G-1, G-4, G-10
Calling
  FORTRAN programs, G-1
  VAX-11 run-time procedures, G-4
  VAX/VMS system services, G-10
CANCEL BREAK, DEBUG command, 9-6
CANCEL MODULE, DEBUG command, 9-3
CANCEL SCOPE, DEBUG command, 9-3
CANCEL TRACE, DEBUG command, 9-7
CANCEL WATCH, DEBUG command, 9-8
CARD-READER, 6-27
Category, data, 3-1
Characters, special, 3-3
/CL:n, file specification switch,
  6-22
Class
  data, 3-1
  test, 4-7
/CO:n, file specification switch,
  6-22
COB$CARDREADER, 6-27
COB$CONSOLE, 6-27
COB$INPUT, 6-28
COB$LINEPRINTER, 6-27
COB$OUTPUT, 6-28, 6-29
COB$PAPERTAPEPUNCH, 6-27
COB$PAPERTAPEREADER, 6-27
COB$SWITCHES, 2-12, 2-13
Command qualifiers, compiler, 2-3
COMP, 4-1
COMP-3, 4-2, 4-4, 6-24
  signs, 4-3
COMP items in a table, 5-4
Comparison, 3-6
Compiler command
  line, 2-2
  qualifiers, 2-3
Compiler error, 10-3, F-1
Computation, optimizing, 13-9
COMPUTATIONAL. See COMP
COMPUTATIONAL-3. See COMP-3
COMPUTE, 4-20
Condition-names, 7-7, 7-8
CONSOLE, 6-27
Constant, figurative, 4-9
CONTINUE command, 2-13
Conventional format, 2-1, 2-3,
  7-1, 8-1
COPY, 2-3, 12-1, 12-2
COPY REPLACING, 12-4
/COPY LIST, 2-3
CORRESPONDING, 3-13
COUNT, in STRING, 3-30
CREATE command, 2-2
/CROSS_REFERENCE, 2-3
/CROSS_REFRENCE, LINK qualifier,
  2-9
CTRL/Y, 2-13
  in DEBUG, 9-9
Current record area, 6-10
Cursor addressing, G-6

Data
  category, 3-1
  class, 3-1
  movement, 3-7
Data Division size limitation,
  B-1
Data item
  elementary, 3-2
  group, 3-2
Data-name limitation, B-1
DEASSIGN command, 2-13
/DEBUG, 2-4
DEBUG, 9-1
/DEBUG, LINK qualifier, 2-10
/DEBUG, RUN qualifier, 2-13
Debugging, 9-1
Decimal scaling, 4-3
DEFINE command, 2-12, 2-13

DELIMITED BY
  in STRING, 3-16
  in UNSTRING, 3-25
DELIMITER, in UNSTRING, 3-31
DEPOSIT, DEBUG command, 9-10
Device, 6-18
Diagnostic error messages, 10-1
Directory, 6-18
DISPLAY, 4-1, 5-26, 6-27, 6-29
DISPLAY statement limitation, B-1
DIVIDE, 4-19
/DQ:n, file specification switch, 6-22
/DW, file specification switch, 6-22
Dynamic group item, 5-17

Edited move, 3-10, 4-10
Editing, 4-10, 4-11
Elementary
  item, 3-2
  move, 3-9, 4-9
Entering a source program, 2-2
Error
  fatal, 2-7
  I/O, 6-25
  message summary, 2-7
  messages, 10-1
  messages, compile-time, 10-1
  messages, diagnostic, 10-1
  messages, link-time, 10-5
  messages, run-time, 10-5
  procedures, I/O, 10-7
  system, compile-time, 10-3, F-1
ESC character, G-6
Escape sequences, G-6
EXAMINE, DEBUG command, 9-10
Examples, programming, G-1
/EXECUTABLE, LINK qualifier, 2-10
Executing a COBOL image, 2-12
EXIT, DEBUG command, 9-9

Figurative constant, 4-9
File
  attribute, 6-1
  compatibility, 6-24
  design, 13-2
  disk, indexed, 6-9
  disk, relative, 6-8
  disk, sequential, 6-7
  handling, 6-1
  I/O interface, in sorting, 11-2
  indexed, 13-3
  library, 12-2
  magnetic tape, 6-6
  name, 6-18

File, (continued)
  record I/O interface, in sorting, 11-2
  relative, 13-2
  sequential, 13-2
  sharing, 6-23
  specification, 6-17, 6-18, 6-20
  switches, 6-19, 6-22
  type, 6-18
  version, 6-18
Files
  naming, 6-17
  opening, 6-14
Fixed-length record, 6-2
Format
  conventional, 2-1, 2-3, 7-1, B-1
  record, 6-2
  reference, 2-1
  terminal, 2-1, 2-3, 7-1, 8-1
FORTRAN programs
  calling, G-1
  files for, 6-24
/FULL, LINK qualifier, 2-10

General formats, A-1
GIVING, 4-16
GO TO DEPENDING limitation, B-1
GO, DEBUG command, 9-8
Group
  move, 3-9, 4-8
Group item, 3-2
  dynamic, 5-17

I/O
  buffer, 6-13
  error procedures, 10-7
  error processing, 6-25
Image execution, 2-12
/INCLUDE, LINK qualifier, 2-10, 2-11
Index data item, 5-15
Index-name, 5-13, 5-16
  initialization, 5-14
Index roots, caching, 13-8
Indexed file, 6-9, 13-3
Indexes, 5-13
Indexing, 5-9
  relative, 5-14
INSPECT, 3-39
Intermediate results, 4-13, 4-22, 4-23
Interrupting image, in DEBUG, 9-9
INVALID KEY, 6-16, 6-26, 10-7
IS, 7-3

Justified move, 3-11

Key buffer, in sort, 11-4
Key, sort, 11-3

Library
  facility, 12-1
  file, 12-2
/LIBRARY, LINK qualifier, 2-11
Limitations, compiler, B-1
LINAGE, 6-17
LINE-PRINTER, 6-27
LINK command, 2-7, 2-9
LINK qualifiers, 2-9
Link-time error messages, 10-5
Linking, 2-7
/LIST, 2-4
Listing, program, C-1
Literal subscripting, 5-10
/LO, file specification switch, 6-22
Locations
  DEBUG, resolving ambiguities, 9-5
  specifying in DEBUG, 9-4
Logical name, 6-19, 6-20, 6-27

Magnetic tape file, 6-6
/MAP, 2-5, C-2
/MAP, LINK qualifier, 2-10
/MI, file specification switch, 6-23
Mnemonic-name, 6-27
MOVE, 3-8, 4-8
Move
  edited, 3-10, 4-10
  elementary, 3-9, 4-9
  group, 3-9, 4-8
  justified, 3-11
  subscripted, 3-12
MOVE CORRESPONDING, 3-13
Multiple delimiters in UNSTRING, 3-29
Multiple operands, 4-16
Multiple receiving fields, 3-12
MULTIPLY, 4-18

/NOANSI_FORMAT, 2-3
/NOCOPY_LIST, 2-3
/NOCROSS_REFERENCE, 2-3
Node, 6-18
/NODEBUG, 2-4
/NODEBUG, RUN qualifier, 2-13
/NOLIST, 2-4
/NOMAP, 2-5
/NOOBJECT, 2-6
/NOVERB_LOCATION, 2-6
/NOWARNINGS, 2-6
Numeric class, 3-1
Numeric data, 4-1

/OBJECT, 2-6
OCCURS, 5-2, 5-3, 5-9, 5-13
Open mode, 6-14
OPEN statement execution, 6-16
Opening files, 6-14
Optimization, 13-1
Optimizing computation, 13-9
OPTIONAL, 6-16
/OPTIONS, LINK qualifier, 2-12
OVERFLOW
  in STRING, 3-18
  in UNSTRING, 3-36

Packed-decimal, 4-2
PAPER-TAPE-PUNCH, 6-27
PAPER-TAPE-READER, 6-27
PERFORM, 7-6
Physical block, 6-5, 13-9
PIC, 7-3
PICTURE, 7-3
POINTER
  in STRING, 3-15
  in UNSTRING, 3-32
PRINT-CONTROL, 6-17
Procedure-name limitation, B-1
Program
  listing, C-1
  organization, 13-8
Programming examples, G-1
Punctuation, use of, 7-5

Qualification, 7-9
  limitation, B-1

Receiving fields, multiple, 3-12
Record
  area, current, 6-10
  areas, sharing, 6-11
  blocking, 6-5
  size, 6-4
Record format, 6-2
  fixed-length, 6-2
  variable-length, 6-3
  VFC, 6-3
Record Management Services, 6-1
Record Unit Size, 6-5
Reference format, 2-1, 7-1, 8-1
REFORMAT, 8-1
Relation test, 3-4, 4-6
Relative file, 6-8, 13-2
Relative indexing, 5-14
RESERVE, 6-13
Restarting program, in DEBUG, 9-8
RMS, 6-1
  buffer defaults, 6-13

ROUNDED, 4-14
RTS, 2-8
RUN command, 2-13
Run-time
  error messages, 10-5
  procedures, calling, G-4
  System, 2-8

SAME AREA, 6-14, 6-16
SAME RECORD AREA, 6-11
Scaling, decimal, 4-3
Scope, DEBUG, 9-3
SEARCH, 5-17, 5-18, 5-19
SELECT, 6-20
Sequential
  disk file, 6-7
  file, 13-2
SET, 5-16
SET BREAK, DEBUG command, 9-6
SET LANGUAGE, DEBUG command, 9-2
SET MODULE, DEBUG command, 9-2
SET SCOPE, DEBUG command, 9-3
SET TRACE, DEBUG command, 9-6
SET WATCH, DEBUG command, 9-7
/SH, file specification switch, 6-23
Sharing
  buffers, 6-14
  record areas, 6-11
SHOW BREAK, DEBUG command, 9-6
SHOW CALLS, DEBUG command, 9-9
SHOW LOGICAL command, 2-13
SHOW MODULE, DEBUG command, 9-3
SHOW SCOPE, DEBUG command, 9-3
SHOW TRACE, DEBUG command, 9-7
SHOW WATCH, DEBUG command, 9-8
Sign test, 4-7
Signs, 4-4
  COMP-3, 4-3
SIZE ERROR, 4-15
Software Performance Report, F-2
Sort keys, 11-3
Sort subroutines, 11-6
Sorting, 11-1
  file I/O interface, 11-2
  record I/O interface, 11-2
Source listing, C-1
Special characters, 3-3
SPECIAL-NAMES, 6-27
SPR, F-2
Statements
  arithmetic, 4-13
STEP, DEBUG command, 9-9
STOP literal, 2-13
  in DEBUG, 9-9
STRING, 3-14
Subroutines, sort, 11-6

Subscripted fields
  in INSPECT, 3-47
  in STRING, 3-20
  in UNSTRING, 3-37
Subscripted move, 3-12
Subscripting, 5-9
  data-name, 5-12
  indexes, 5-13
  literal, 5-10
SUBTRACT, 4-18
  multiple operands, 4-16
Switches
  file specification, 6-19, 6-22
  program, 2-12, 2-13
SYNCHRONIZED, 5-4
SYS$LIBRARY:C74LIB/LIB, 2-8
SYS$LIBRARY:C74LIB/OPT, 2-8
System error, compile-time, 10-3, F-1
System messages, VAX/VMS, 10-4
System services, calling, G-10

Table
  handling, 5-1
Tables
  COMP items in, 5-4
  defining, 5-1
  initializing, 5-7
  variable-length, 5-17
TALLYING
  in INSPECT, 3-48
  in UNSTRING, 3-34
Tape file, 6-6
Temporary work area
  arithmetic expression, 4-21
Terminal escape sequences, G-6
Terminal format, 2-1, 2-3, 7-1, 8-1
Test
  class, 4-7
  relation, 3-4, 4-6
  sign, 4-7
Traceback, 2-4, 2-10, 9-9, 10-6, 10-8
/TRACEBACK, LINK qualifier, 2-10
Tracepoint, DEBUG, 9-6, 9-7

UNSTRING, 3-22
USE procedure, 6-26

VALUE, 5-7
VALUE OF ID, 6-20
Variable-length
  record, 6-3
  table, 5-17
Variable with fixed-length control record. See VFC

VAX-11 run-time procedures,
  calling, G-4
VAX-11 Sort, 11-1
VAX/VMS system messages, 10-4
VAX/VMS system services, calling,
  G-10
/VERB_LOCATION, 2-6, C-2
Version, file, 6-18
VFC, 6-29
VFC record, 6-3

/WARNINGS, 2-6
Watchpoint, DEBUG, 9-7, 9-8
/WI:n, file specification switch,
  6-23

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify)_____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or
Country

—Do Not Tear - Fold Here and Tape —

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN: Commercial Engineering Publications MK1-2/H3
DIGITAL EQUIPMENT CORPORATION
CONTINENTAL BOULEVARD
MERRIMACK N.H. 03054

Do Not Tear - Fold Here and Tape

Cut Along Dotted Line

digital

PRINTED IN USA